Technical Report

UCAM-CL-TR-834
ISSN 1476-2986

Number 834

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Concurrent verification for sequential programs

John Wickerson

May 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2012 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Concurrent verification for sequential programs

John Wickerson

Summary

This dissertation makes two contributions to the field of software verification. The first explains how verification techniques originally developed for concurrency can be usefully applied to sequential programs. The second describes how sequential programs can be verified using diagrams that have a parallel nature.

The first contribution involves a new treatment of stability in verification methods based on rely-guarantee. When an assertion made in one thread of a concurrent system cannot be invalidated by the actions of other threads, that assertion is said to be 'stable'. Stability is normally enforced through side-conditions on rely-guarantee proof rules. This dissertation proposes instead to encode stability information into the syntactic form of the assertion. This approach, which we call **explicit stabilisation**, brings several benefits. First, we empower rely-guarantee with the ability to reason about library code for the first time. Second, when the rely-guarantee method is redeployed in a sequential setting, explicit stabilisation allows more details of a module's implementation to be hidden when verifying clients. Third, explicit stabilisation brings a more nuanced understanding of the important issue of stability in concurrent and sequential verification; such an understanding grows ever more important as verification techniques grow ever more complex.

The second contribution is a new method of presenting program proofs conducted in separation logic. Building on work by Jules Bean, the **ribbon proof** is a diagrammatic alternative to the standard 'proof outline'. By emphasising the structure of a proof, ribbon proofs are intelligible and hence useful pedagogically. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they are highly scalable; this we illustrate with a ribbon proof of the Version 7 Unix memory manager. Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs. We describe the ribbon proof system, prove its soundness and completeness, and outline a prototype tool for mechanically checking the diagrams it produces.

Acknowledgements

Matthew Parkinson, my primary supervisor, and latterly my industrial supervisor, has been wonderful throughout, and I thank him enormously. He has given me both the freedom to explore my own lines of research and the guidance to keep me from the wilder shores of speculation. He has been consistently full of brilliant ideas and sage advice, and always generous with his time. Dear reader: if you ever have the opportunity to study for a PhD with Matthew, I strongly advise that you take it.

I thank Glynn Winskel, latterly my primary supervisor, for nurturing my interest in the theoretical side of computer science, chiefly through his fascinating Discrete Maths lectures; Mike Dodds, my secondary supervisor, for being a great collaborator, for hundreds of lively discussions, and for his friendship; and Sir Tony Hoare for hosting my internship at Microsoft Research Cambridge, and for continuing to be an inspiring mentor.

I also thank my examiners, Cliff Jones and Mike Gordon, for their support and encouragement; my office mates Matko Botinčan and Eric Koskinen for being such pleasant company, and for graciously tolerating my VERY NOISY TYPING; Mike Phelan for making computer science appealingly easy, and John Fawcett for making it appealingly difficult – both were inspiring teachers; Richard Bornat for infecting me with his contagious enthusiasm, and for suggesting the Version 7 Unix memory manager as a verification target; Joey Coleman for his support in the early days of my PhD studies, and for inspiring the picture on page 51; Rustan Leino for pointing out that my explicit stabilisation operators form a Galois connection; Rasmus Petersen for drawing the pictures on page 102, and suggesting rounding the corners of if- and while-blocks in ribbon proofs; Tom Ridge for his constructive scepticism and handy emacs macros; Noam Rinetzky for daring to suggest a 'guitar hero' interface for ribbon proofs; Hongseok Yang for the original idea of parameterising rely-guarantee specifications by the 'current rely'; Lise Gough, Carol Nightingale and Tanya Hall for dispatching all manner of administrative obstacles with aplomb; the people behind Aquamacs, BibDesk, LaTeX and Skim, for easing my dissertation-writing process; Mark Batty, Jules Bean, Nick Benton, Ernie Cohen, Miklós Danka, Thomas Dinsdale-Young, Sophia Drossopoulou, Xinyu Feng, Philippa Gardner, Jonathan Hayman, Aquinas Hobor, Alexander Malkis, Andrew Pitts, Thomas Santen, Peter Sewell, Sam Staton, Alexander Summers, Stephan Tobies, Mark Wheelhouse, and the anonymous reviewers of ESOP 2010, LICS 2012 and ESOP 2013 for helpful comments and encouragement; and the the super Ben Hanks for his assiduous proofreading.

I gratefully acknowledgement financial support from EPSRC grant F019394/1 and a scholarship from Churchill College.

Finally, I thank my parents for giving me the best possible start and for their amazing support, and I thank my wonderful wife Erica, for being there for me every step of the way, and whose love makes it all worthwhile.

To my wife and best friend.

Contents

List of Figures

Chapter 1

Introduction

It is a truth universally acknowledged that any sizeable piece of software will contain errors. Much effort goes into testing software before it is released, but bugs still remain. McConnell [2004] reports an industry average, in delivered programs, of between 1 and 25 errors per thousand lines of code, while Tennent [2002] puts it at between 10 and 17. The effects of these errors can be catastrophic, one well-known example being the loss of the European Space Agency's first *Ariane 5* rocket, thirty-eight seconds after launch, as a result of a simple programming error that was not spotted during testing [Lacan et al. 1998]. Even mundane errors can be hugely damaging: a report by the U.S. Department of Commerce [NIST 2002] estimated that the economic costs of faulty software "range in the tens of billions of dollars per year" or "just under one percent of the nation's gross domestic product (GDP)."

This dissertation is concerned with an alternative to software testing called software verification. Observing that mathematicians did not confirm Fermat's last theorem by trying it on lots of numbers, software verification aims not to *discover* bugs but to construct watertight arguments that *no bugs exist*. The first formal system for verifying programs was described by Hoare [1969], and since then the topic has been studied intensively in academia. It is a great effort to translate a program, and what it means for that program to be correct, into formal mathematics. As such, verification remains relatively rarely used in the software industry, except in such safety-critical projects as the control system of the next *Ariane 5* rocket.

Nevertheless, the arguments for software verification over testing are becoming increasingly persuasive. Verification may be a significant effort, but so is testing, which typically consumes "approximately 50 percent of the elapsed time and more than 50 percent of the total cost" of the overall development process [Myers et al. 2012]. Concurrent programs are becoming the norm in the software industry, but can harbour subtle bugs whose effects are very hard to reproduce during testing. And with verification tools for both sequential and concurrent programs becoming increasingly powerful and user-friendly, verification is gradually becoming a feasible option outside of the safety-critical arena.

We should clarify that verified software is not necessarily perfect software. If the software is verified by a human, they may have made a mistake, and if the software is verified by a tool, then the human who created the tool may have made a mistake. A program can only be verified relative to a given specification, which may not accurately capture the intended requirements of the program. The compiler that translates the verified program into machine code may introduce errors (although work on verified compilers, as pioneered by McCarthy and Painter [1967], is ongoing). The hardware that executes this machine code may be faulty (although work on verified hardware is ongoing; one prominent early application being the verification of a microprocessor by Hunt [1985]). Even if each of these stages can be verified, the physi-

cal environment in which the software runs can never be entirely specified in mathematics: a freak electrical surge might cause a bit to flip, an earthquake might cause a hard drive to crash. Despite these imperfections, it turns out in practice that verified software is far more reliable than non-verified software. Holzmann [2001] has found that the "application of software verification [...] can increase the number of software defects intercepted during system testing ten-fold, when compared with conventional testing," while programming teams at NASA have used mathematical specification and formal methods to bring the error rate in their space shuttle control system down to just 0.002 errors per thousand lines of code [Fishman 1996].

This dissertation makes two contributions to the field of software verification, both under the banner of 'concurrent verification for sequential programs'. The two-part thesis is stated as follows.

> **Thesis.** (1) Techniques originally developed for the verification of concurrent programs can be usefully applied to the verification of sequential modules that expose to their clients some information about their internal state. (2) Diagrams that appear to parallelise sequential programs can be usefully applied to verifying such programs.

We tackle the first part of this thesis by describing, in Chapter 4, how RGSep – a verification technique developed for concurrent programs – can be adapted to the task of verifying sequential modules. The reader may suspect that this task is trivial, since a sequential program is merely a degenerate concurrent one. We are concerned, however, with the verification of sequential *modules*, which turns out to raise similar challenges to the verification of concurrency. The link between sequential modules and concurrent programs is well-known, having been described by Vafeiadis [2007] among others. What is new here are insights about how to verify a module that allows its clients to track certain aspects of its internal state.

An important technical prerequisite for the application of RGSep to sequential modules is a technique called *explicit stabilisation*, which is introduced in Chapter 3. That chapter also explains various other benefits that explicit stabilisation can bring to concurrency verification; in particular, the first modular version of Jones' popular rely-guarantee method.

The second part of the thesis concerns the application of diagrams to program verification, and is the topic of Chapter 5. One feature of our diagrams is that they depict sequential programs in a more concurrent manner; that is, without the total order on instructions that sequential programs normally impose. We find that our diagrams can make proofs about programs more readable, easier to modify, and more scalable.

As a recurring case study, we use the memory manager from Version 7 Unix [Bell Labs 1979]. Collaborative work with Mike Dodds and Matthew Parkinson has produced the first formal verification of this program – which dates from 1979 – and uncovered a serious bug while doing so (see Sect. 4.3.4). A memory manager is a program responsible for handling requests for memory from other programs. Such requests could be passed directly to the operating system, but a well-implemented memory manager will normally be able to speed up this process. It usually does so by keeping track of which memory has recently been released, and reusing that whenever possible.

The layman may think of a memory manager as the manager of a large apartment block. The manager, called Bob, oversees short- and long-term rentals by various tenants. The apartments are of various sizes. These sizes are flexible; for instance, one four-room apartment can be easily re-advertised as four one-room apartments by locking the three doors between its rooms. Always seeking to please, Bob will arrange for a new apartment to be built (subject to obtaining planning permission) whenever he is unable to accommodate a new tenant immediately.

The essence of the verification challenge is to prove that no apartment or room is ever 'double booked'. Moreover, the verification must be done in a *modular* fashion. This means, firstly, that Bob's operating procedure must be verified without knowing the behaviour of any current or future tenants, and secondly, that no client should need to know Bob's particular procedure – a contract clarifying his duties should suffice. This enables Bob to vary the details of his procedure without having to notify clients. For instance, it happens that his policy is always to place new tenants in the lowest-numbered sufficiently-large available apartment, but tenants need not know this. Researchers have previously developed techniques to verify apartment blocks like Bob's. O'Hearn et al. [2004] describe one technique, but only consider an apartment block consisting solely of two-room apartments. Such a block is less appealing to larger families, and it is much easier to confirm that no room is ever double-booked. Parkinson and Bierman [2005] describe another technique. Although they do consider arbitrarily-sized apartments, their manager has a somewhat inefficient policy of knocking down vacated apartments and building new ones upon demand. Again, the question of double-booking becomes easily answerable. We discover that the verification of our apartment block, with its variably-sized apartments and its policy of reusing vacated rooms, requires more than a straightforward combination of O'Hearn et al.'s technique with that of Parkinson and Bierman

Outline and contributions

Chapter 2 We survey the software verification techniques upon which later chapters build. We give a unified presentation of all of the techniques, which serves to emphasise the similarities and the essential differences between them.

Chapter 3 We present a software verification technique called explicit stabilisation. We explain how explicit stabilisation enables the simplification of complex rely-guarantee proof rules, the verification of concurrent library code using rely-guarantee, and the unification of several previously-disparate varieties of rely-guarantee reasoning.

Chapter 4 We explain how a concurrent verification technique called RGSep can be usefully applied, with the help of explicit stabilisation, to the verification of sequential modules. We present a proof of the memory safety of the memory manager from Version 7 Unix (having fixed a bug in the code).

Chapter 5 We introduce ribbon proofs: a system of diagrams that present proofs about programs in a more intuitive way than traditional 'proof outlines'. We give proof rules, show them to be sound and complete, and outline a prototype tool for mechanically checking our diagrammatic proofs.

Chapter 6 We conclude, and discuss some potential future work.

Chapter 2

Background

This chapter surveys several techniques in software verification, upon which our later chapters build. We introduce Hoare logic, and its more recent descendant, separation logic. Moving to program logics for concurrency, we describe the Owicki-Gries method, the rely-guarantee method, concurrent separation logic, and RGSep. Figure 2.1 depicts how these techniques build upon each other, and where applicable, the sections of this chapter in which they are described.

2.1 A sequential programming language

In the interests of clarity, we present the results of this dissertation in the context of a very simple (but highly extensible) programming language. Let Command be the set of *commands* in our language. (Later we shall use the word *program* when these commands are structured into procedures or modules.) We populate this set using the following constructions.

$$\begin{array}{rll} C ::= & \texttt{skip} & \text{(do nothing)} \\ \mid & C \texttt{ ; } C & \text{(sequential composition)} \\ \mid & C \texttt{ or } C & \text{(non-deterministic choice)} \\ \mid & \texttt{loop}\, C & \text{(non-deterministic loop)} \\ \mid & c & \text{(basic command)} \end{array}$$

We give, in Fig. 2.2, an operational semantics as a transition relation between configurations. A configuration (C, σ) comprises a command C and a program state $\sigma \in \mathsf{PState}$. For now, we shall leave the definition of PState open. We write $\rightarrow$ to denote a single step of computation. We treat $\mathbf{abort}$ as a special configuration that is reached whenever a fault has occurred. For each basic command c, we define a function

$$[\![c]\!] : \mathsf{PState} \rightarrow (\mathcal{P}(\mathsf{PState})) \uplus \{\mathbf{abort}\}$$

such that if σ is an initial state, $[\![c]\!](\sigma)$ is the set of possible final states that the execution of c might yield, unless execution could cause a fault, in which case $[\![c]\!](\sigma)$ is $\mathbf{abort}$. We shall occasionally wish to summarise a basic command c as a relation between program states; this is done using a function defined as:

$$\mathsf{transitions}(c) \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \sigma' \in [\![c]\!](\sigma)\}.$$

The choice of basic commands is independent of the theory developed in this dissertation. However, in some of our examples, it is helpful to assume that our programming language is fleshed

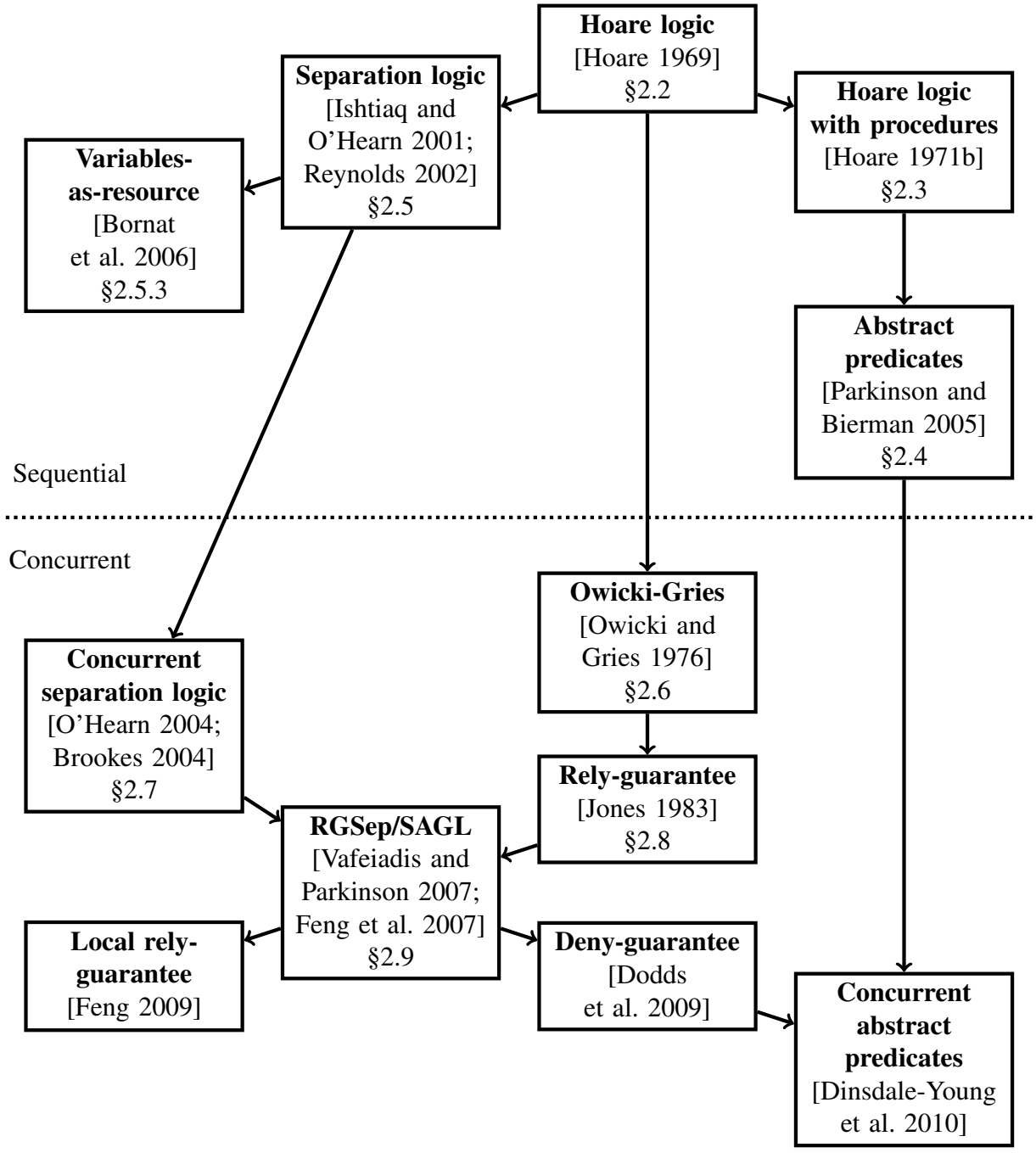


Figure 2.1: A family tree of selected program logics

$$\frac{}{(\texttt{skip}\ ;\ C, \sigma) \to (C, \sigma)} \qquad \frac{(C_1, \sigma) \to (C_1', \sigma')}{(C_1\ ;\ C_2, \sigma) \to (C_1'\ ;\ C_2, \sigma')} \qquad \frac{(C_1, \sigma) \to \mathbf{abort}}{(C_1\ ;\ C_2, \sigma) \to \mathbf{abort}}$$

$$\frac{}{(C_1\ \texttt{or}\ C_2, \sigma) \to (C_1, \sigma)} \qquad \frac{}{(C_1\ \texttt{or}\ C_2, \sigma) \to (C_2, \sigma)}$$

$$\frac{}{(\texttt{loop}\, C, \sigma) \to (\texttt{skip}\ \texttt{or}\ (C\ ;\ \texttt{loop}\, C), \sigma)} \qquad \frac{\sigma' \in [\![c]\!](\sigma)}{(c, \sigma) \to (\texttt{skip}, \sigma')} \qquad \frac{[\![c]\!](\sigma) = \mathbf{abort}}{(c, \sigma) \to \mathbf{abort}}$$

Figure 2.2: Operational semantics

out with some basic commands, such as assumptions, run-time assertions, and assignments. To this end, let E range over the set ProgExp of program expressions, and let B range over the subset BoolProgExp comprising boolean-valued program expressions. Then:

$$\begin{array}{rll} c ::= & \texttt{assume}\, B & \text{(assumption)} \\ \mid & \texttt{assert}\, B & \text{(run-time assertion)} \\ \mid & x := E & \text{(assignment)} \end{array}$$

Both $\texttt{assume}(B)$ and $\texttt{assert}(B)$ do nothing if the expression B holds, but while the latter causes a fault when B does not hold, the former just gets stuck. We shall write $[\![B]\!](\sigma)$ or $[\![E]\!](\sigma)$ for the result of evaluating (instantaneously) an expression in a program state σ.

$$[\![\texttt{assume}\, B]\!](\sigma) \stackrel{\text{def}}{=} \begin{cases} \{\sigma\} & \text{if } [\![B]\!](\sigma) = \mathit{true} \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\texttt{assert}\, B]\!](\sigma) \stackrel{\text{def}}{=} \begin{cases} \{\sigma\} & \text{if } [\![B]\!](\sigma) = \mathit{true} \\ \mathbf{abort} & \text{otherwise} \end{cases}$$

By combining assumptions with our non-deterministic choice and loop commands, we can encode the more practical `if` and `while` commands as follows:

$$\begin{aligned} \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 &\stackrel{\text{def}}{=} (\texttt{assume}\, B\ ;\ C_1)\ \texttt{or}\ (\texttt{assume}\, \neg B\ ;\ C_2) \\ \texttt{while}\, B\ \texttt{do}\, C &\stackrel{\text{def}}{=} \texttt{loop}(\texttt{assume}\, B\ ;\ C)\ ;\ \texttt{assume}\, \neg B. \end{aligned}$$

To describe the effect of an assignment command, we must specify the nature of program states. A program state σ is a mapping from program variables $x \in \mathsf{PVar}$ to values v; that is:

$$\mathsf{PState} \stackrel{\text{def}}{=} \mathsf{PVar} \to \mathsf{Val}$$

We shall write $\sigma[x \mapsto v]$ for the program state that is the same as σ but with x updated to value v. The semantics of assignment is then:

$$[\![x := E]\!](\sigma) \stackrel{\text{def}}{=} \{\sigma[x \mapsto v]\} \quad \text{where } v = [\![E]\!](\sigma).$$

2.2 Hoare logic

Hoare logic [Hoare 1969] is a system for making judgements about the behaviour of programs. The behaviour of a program is specified as a *Hoare triple*, which is written

$$\{p\}\ C\ \{q\}$$

and comprises a *precondition* p, a command C and a *postcondition* q. The precondition and postcondition together constitute the *specification* of C. We write

$$\models_{\mathsf{HL}} \{p\}\ C\ \{q\}$$

to mean that the triple $\{p\}\ C\ \{q\}$ is valid. A triple is valid if whenever execution of the command C begins in a state satisfying the precondition p, then execution will not fault, and if it terminates, it will do so in a state satisfying the postcondition q. This interpretation is called *partial correctness* because it does not constrain the behaviour of programs that do not terminate. The clause that 'execution will not fault' does not appear in traditional definitions of Hoare logic. We include it partly in preparation for the upcoming introduction of separation logic.

The precondition and postcondition are drawn from a set $\mathsf{Assertion}$ of *assertions*. We introduce a set LVar of logical variables, disjoint from program variables. These aid the expressivity of assertions. We define a logical state $(\sigma, i) \in \mathsf{LState}$ to be a program state σ augmented with an *interpretation* $i \in \mathsf{LVar} \to \mathsf{Val}$ mapping logical variables to their values. That is, we define

$$\mathsf{LState} \stackrel{\text{def}}{=} \mathsf{PState} \times (\mathsf{LVar} \to \mathsf{Val}).$$

Just as E ranges over the set $\mathsf{ProgExp}$ of expressions over program states, let e range over a set Exp of expressions over logical states. The value of expression e in logical state (σ, i) is written $[\![e]\!](\sigma, i)$.

We populate the set of assertions using the following constructions.

$$\begin{array}{ll} p ::= \mathit{true} \mid \mathit{false} & \text{(boolean constants)} \\ \quad \mid p \wedge p & \text{(conjunction)} \\ \quad \mid p \vee p & \text{(disjunction)} \\ \quad \mid p \Rightarrow p & \text{(implication)} \\ \quad \mid \exists x.\, p & \text{(existential quantification over a logical variable)} \end{array}$$

If p is an assertion, we write $[\![p]\!]$ for the set of logical states that satisfy it.

$$\begin{array}{rcl} [\![\mathit{true}]\!] & = & \mathsf{LState} \\ [\![\mathit{false}]\!] & = & \emptyset \\ [\![p \wedge q]\!] & = & [\![p]\!] \cap [\![q]\!] \\ [\![p \vee q]\!] & = & [\![p]\!] \cup [\![q]\!] \\ [\![p \Rightarrow q]\!] & = & [\![p]\!]^{\mathsf{c}} \cup [\![q]\!] \\ [\![\exists x.\, p]\!] & = & \{(\sigma, i) \mid \exists v.\, (\sigma, i[x \mapsto v]) \in [\![p]\!]\} \end{array}$$

We can encode our intuitive description of the meaning of a Hoare triple $\{p\}\ C\ \{q\}$ formally and concisely, in the spirit of Vafeiadis [2011], in the definition below. We use a greatest fixpoint construction to capture, in the set $\mathit{safe}_{\mathsf{HL}}(q)$, all configurations from which no execution ends in $\mathbf{abort}$, and from which all terminating executions end in a configuration whose state satisfies q. We then require that each configuration whose command is C and whose state satisfies p, is in $\mathit{safe}_{\mathsf{HL}}(q)$.

Definition 2.1 (Meaning of judgements: Hoare logic)**.** Let $\mathit{safe}_{\mathsf{HL}}(q)$ be the largest set containing only those triples (C, σ, i) that:

- do not fault:
$$(C, \sigma) \not\rightarrow \mathbf{abort},$$
- satisfy the postcondition q if they are terminal:
$$C = \mathtt{skip} \Longrightarrow (\sigma, i) \in [\![q]\!],$$
- and continue to satisfy these properties after any execution step:
$$\forall C', \sigma'.\, (C, \sigma) \rightarrow (C', \sigma') \Longrightarrow (C', \sigma', i) \in \mathit{safe}_{\mathsf{HL}}(q).$$

We can then define:

$$\models_{\mathsf{HL}} \{p\}\ C\ \{q\} \stackrel{\text{def}}{=} \forall (\sigma, i) \in [\![p]\!].\, (C, \sigma, i) \in \mathit{safe}_{\mathsf{HL}}(q).$$

Remark 2.2. There is an alternative formulation called *total correctness*, which deems Hoare triples to be valid only if the command always terminates. We shall focus only on partial correctness in this dissertation. The semantics of total correctness can be obtained by defining $\mathit{safe}(q)$ as the least fix-point rather than the greatest.

Hoare logic comprises a collection of rules that allow judgements to be made about the validity of Hoare triples. Let us write

$$\vdash_{\mathsf{HL}} \{p\}\ C\ \{q\}$$

when the triple $\{p\}\ C\ \{q\}$ can be deemed valid through the application of these rules. The rules can be adapted according to context, subject to the condition that any judgement they make must indeed be valid; that is, that

$$\vdash_{\mathsf{HL}} \{p\}\ C\ \{q\} \text{ implies } \models_{\mathsf{HL}} \{p\}\ C\ \{q\}\,.$$

A collection of Hoare logic rules used in this dissertation is given in Fig. 2.3. In the CONSEQUENCE rule, the implications must hold in all states; in general, we write $\models p$ to mean $[\![p]\!] = \mathsf{LState}$.

2.2.1 Logical variables and two-state postconditions

We remarked (page 20) that logical variables aid the expressivity of assertions. This is exemplified by the following specification of a command for incrementing a variable:

$$\{\mathtt{x} = X\}\ \mathtt{x := x} + 1\ \{\mathtt{x} = X + 1\}\,.$$

Jones [1990] proposes a different form of Hoare triple, in which the postcondition is a predicate of not one, but two states: those states before and after execution of the command. His system can specify the increment command without needing a logical variable to cache the initial value of x:

$$\{\mathit{true}\}\ \mathtt{x := x} + 1\ \{\mathtt{x} = \overleftarrow{\mathtt{x}} + 1\}\,.$$

Expressions are by default evaluated in the 'after' state, but those with a hook above them are evaluated in the 'before' state. That is, for expressions e:

$$[\![e]\!]_{2\text{state}}(\sigma, \sigma') \stackrel{\text{def}}{=} [\![e]\!](\sigma')$$
$$[\![\overleftarrow{e}]\!]_{2\text{state}}(\sigma, \sigma') \stackrel{\text{def}}{=} [\![e]\!](\sigma).$$

$$\textsc{Disj}\ \frac{\vdash_{\mathsf{HL}} \{p_1\}\ C\ \{q_1\} \quad \vdash_{\mathsf{HL}} \{p_2\}\ C\ \{q_2\}}{\vdash_{\mathsf{HL}} \{p_1 \vee p_2\}\ C\ \{q_1 \vee q_2\}} \qquad \textsc{Conj}\ \frac{\vdash_{\mathsf{HL}} \{p_1\}\ C\ \{q_1\} \quad \vdash_{\mathsf{HL}} \{p_2\}\ C\ \{q_2\}}{\vdash_{\mathsf{HL}} \{p_1 \wedge p_2\}\ C\ \{q_1 \wedge q_2\}}$$

$$\textsc{Consequence}\ \frac{\vdash_{\mathsf{HL}} \{p'\}\ C\ \{q'\} \quad \models (p \Rightarrow p') \quad \models (q' \Rightarrow q)}{\vdash_{\mathsf{HL}} \{p\}\ C\ \{q\}}$$

$$\textsc{Exists}\ \frac{\vdash_{\mathsf{HL}} \{p\}\ C\ \{q\}}{\vdash_{\mathsf{HL}} \{\exists x.\, p\}\ C\ \{\exists x.\, q\}} \qquad \textsc{Skip}\ \frac{}{\vdash_{\mathsf{HL}} \{p\}\ \mathtt{skip}\ \{p\}} \qquad \textsc{Seq}\ \frac{\vdash_{\mathsf{HL}} \{p\}\ C_1\ \{r\} \quad \vdash_{\mathsf{HL}} \{r\}\ C_2\ \{q\}}{\vdash_{\mathsf{HL}} \{p\}\ C_1\ ;\ C_2\ \{q\}}$$

$$\textsc{Choice}\ \frac{\vdash_{\mathsf{HL}} \{p\}\ C_1\ \{q\} \quad \vdash_{\mathsf{HL}} \{p\}\ C_2\ \{q\}}{\vdash_{\mathsf{HL}} \{p\}\ C_1\ \mathtt{or}\ C_2\ \{q\}} \qquad \textsc{Loop}\ \frac{\vdash_{\mathsf{HL}} \{p\}\ C\ \{p\}}{\vdash_{\mathsf{HL}} \{p\}\ \mathtt{loop}\, C\ \{p\}} \qquad \textsc{Assume}\ \frac{}{\vdash_{\mathsf{HL}} \{p\}\ \mathtt{assume}\, B\ \{p \wedge B\}}$$

$$\textsc{Assert}\ \frac{}{\vdash_{\mathsf{HL}} \{p \wedge B\}\ \mathtt{assert}\, B\ \{p\}} \qquad \textsc{Assign}\ \frac{}{\vdash_{\mathsf{HL}} \{p[E/x]\}\ x := E\ \{p\}}$$

Figure 2.3: Proof rules for Hoare Logic

A disadvantage of two-state postconditions is that the proof rules become a little more complex, having lost the appealing symmetry between the pre- and postconditions. Hence, in the interests of theoretical simplicity, and in keeping with much of the recent literature on program verification, we shall use single-state postconditions throughout this dissertation. Nevertheless, we shall find the hook notation useful for defining rely and guarantee relations; moreover, the following generalisation is useful for lifting a set to a relation whose preimage is that set:

$$\overleftarrow{S} \stackrel{\text{def}}{=} \{(x, y) \mid x \in S\}.$$

2.3 Simple procedures and modules

This section describes how to extend Hoare logic to reason about programs that are structured into simple procedures or modules.

We now extend our programming language with procedures. Let k range over a set $\mathbb{K}$ of procedure names. Suppose each procedure name is associated with an arity: the number of arguments it takes. When procedures are called or defined, we shall tacitly assume that the list of arguments is of the correct length. We introduce to the syntax of commands a construction for procedure calls

$$\begin{aligned} C ::= {} & \ldots \\ & \mid y := k(E, \ldots, E) \qquad \text{(procedure call)} \end{aligned}$$

and we define a procedure dictionary $\eta \in \mathsf{ProcDict}$ to be a mapping from a finite set of procedure names to a list of formal arguments and a body:

$$\mathsf{ProcDict} \stackrel{\text{def}}{=} \mathbb{K} \rightharpoonup_{\text{fin}} \mathsf{PVar} \times \ldots \times \mathsf{PVar} \times \mathsf{Command}.$$

We shall notate elements of ProcDict as follows:

$$k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n.$$

A complete program, written $\texttt{procs}\ \eta\ \texttt{in}\ C$, comprises a procedure dictionary η and a 'main' routine C that may call procedures defined in η. We use *prog* to range over complete programs.

We impose that procedure bodies do not contain calls. This rather draconian condition ensures that our programming language is not complicated by recursive calls, and hence makes our theory considerably simpler. It still allows us to express our memory manager case study, which comprises just two non-recursive procedures: `malloc` and `free`.

Another simplifying assumption is that the set of variables accessed by the procedure bodies is disjoint from the set of variables accessed by the main routine. Data can still be communicated to the procedures via the call-by-value arguments, and back to the caller via the return value. Later, when we augment the state with a heap component, data will also be transferable via the heap. The variable-disjointness restriction could be enforced simply by requiring that all of the variables and formal parameters in the procedure bodies – and none of the variables in the main routine – are given a special prefix such as '`private_`'. We would also require, during verification, that assertions in the procedure bodies do not refer to non-private variables, and that assertions in the main routine do not refer to private variables. Having noted that this solution exists, we shall henceforth ignore the problem, and name all variables freely.

The operational semantics of programs is as follows. A procedure call fails if the procedure name is not in the dictionary. Otherwise, the actual parameters $(E_1, \ldots, E_n)$ are assigned to the formal parameters $(x_1, \ldots, x_n)$, then the body is executed, and finally the return value (held in a dedicated program variable `ret`) is assigned to the variable nominated by the caller. We define procedure calls to happen atomically. This makes our semantics unsuitable for reasoning about procedures that execute in a concurrent context, but simplifies some of our proof rules later on.

$$\frac{\eta(k) = (x_1, \ldots, x_n, C) \qquad (C, \sigma[x_1 \mapsto [\![E_1]\!](\sigma), \ldots, x_n \mapsto [\![E_n]\!](\sigma)]) \rightarrow_\eta^* (\texttt{skip}, \sigma')}{(y := k(E_1, \ldots, E_n), \sigma) \rightarrow_\eta (\texttt{skip}, \sigma'[\texttt{ret} \mapsto \sigma'(y)])}$$

$$\frac{\eta(k) = (x_1, \ldots, x_n, C) \qquad (C, \sigma[x_1 \mapsto [\![E_1]\!](\sigma), \ldots, x_n \mapsto [\![E_n]\!](\sigma)]) \rightarrow_\eta^* \mathbf{abort}}{(y := k(E_1, \ldots, E_n), \sigma) \rightarrow_\eta \mathbf{abort}}$$

$$\frac{k \notin \mathrm{dom}(\eta)}{(y := k(E_1, \ldots, E_n), \sigma) \rightarrow_\eta \mathbf{abort}}$$

For complete programs, there are just two rules:

$$\frac{(C, \sigma) \rightarrow_\eta (C', \sigma')}{(\texttt{procs}\ \eta\ \texttt{in}\ C, \sigma) \rightarrow (\texttt{procs}\ \eta\ \texttt{in}\ C', \sigma')} \qquad \frac{(C, \sigma) \rightarrow_\eta \mathbf{abort}}{(\texttt{procs}\ \eta\ \texttt{in}\ C, \sigma) \rightarrow \mathbf{abort}}$$

We can extend Hoare logic to handle procedures by parameterising judgements with a *procedure specification dictionary* $\Gamma \in \mathsf{ProcSpecDict}$. This is a mapping from a finite set of procedure names to a list of formal arguments, a precondition and a postcondition:

$$\mathsf{ProcSpecDict} \stackrel{\text{def}}{=} \mathbb{K} \rightharpoonup_{\text{fin}} \mathsf{PVar} \times \ldots \times \mathsf{PVar} \times \mathsf{Assertion} \times \mathsf{Assertion}.$$

We shall notate elements of ProcSpecDict as follows:

$$\left\{p_1\right\}\ k_1(\bar{x}_1)\ \left\{q_1\right\}, \ldots, \left\{p_n\right\}\ k_n(\bar{x}_n)\ \left\{q_n\right\}.$$

Definition 2.3 (Meaning of judgements: Hoare logic with procedures)**.** Let $safe_{\mathsf{HLP}}(q, \eta)$ be the largest set containing only those triples (C, σ, i) that:

- do not fault:
$$(C, \sigma) \not\to_\eta \mathbf{abort},$$
- satisfy the postcondition q if they are terminal:
$$C = \mathtt{skip} \implies (\sigma, i) \in [\![q]\!],$$
- and continue to satisfy these properties after any execution step:
$$\forall C', \sigma'.\, (C, \sigma) \to_\eta (C', \sigma') \implies (C', \sigma', i) \in safe_{\mathsf{HLP}}(q, \eta).$$

This is the same as in Defn. 2.1, but with an additional parameter η. We can then define an intermediate Hoare logic judgement that is also parameterised by η:

$$\models_\eta^{\mathsf{HLP}} \{p\}\ C\ \{q\} \quad \stackrel{\text{def}}{=} \quad \forall (\sigma, i) \in [\![p]\!].\, (C, \sigma, i) \in safe_{\mathsf{HLP}}(q, \eta).$$

This allows us to define a judgement that is parameterised not by the implementations of procedures, but by their specifications:

$$\Gamma \models^{\mathsf{HLP}} \{p\}\ C\ \{q\} \quad \stackrel{\text{def}}{=} \quad \forall \eta.\, \eta\ \mathsf{implements}_{\mathsf{HL}}\ \Gamma \implies \models_\eta^{\mathsf{HLP}} \{p\}\ C\ \{q\}$$

where procedure implementations (in η) and procedure specifications (in Γ) are related by the following judgement:

$$\eta\ \mathsf{implements}_{\mathsf{HL}}\ \Gamma \quad \stackrel{\text{def}}{=} \quad \forall \{p\}\ k(\bar{x})\ \{q\} \in \Gamma.\, \exists C.\, \eta(k) = (\bar{x}, C) \wedge \models_\eta^{\mathsf{HLP}} \{p\}\ C\ \{q\}.$$

The judgement presented above concerns the behaviour of commands. We require a new form of judgement to handle complete programs.

Definition 2.4 (Meaning of judgements: Hoare logic with procedures, for complete programs)**.** Let $safe_{\mathsf{HLP}}(q)$ be the largest set containing only those triples $(prog, \sigma, i)$ that:

- do not fault:
$$(prog, \sigma) \not\to \mathbf{abort},$$
- satisfy the postcondition q if they are terminal:
$$(\exists \eta.\, prog = \mathtt{procs}\ \eta\ \mathtt{in}\ \mathtt{skip}) \implies (\sigma, i) \in [\![q]\!],$$
- and continue to satisfy these properties after any execution step:
$$\forall prog', \sigma'.\, (prog, \sigma) \to (prog', \sigma') \implies (prog', \sigma', i) \in safe_{\mathsf{HLP}}(q).$$

This is similar to Defn. 2.1, except that the terminating command is no longer `skip`, but `procs` η `in skip` for some η. We can then define:

$$\models^{\mathsf{HLP}} \{p\}\ prog\ \{q\} \quad \stackrel{\text{def}}{=} \quad \forall (\sigma, i) \in [\![p]\!].\, (prog, \sigma, i) \in safe_{\mathsf{HLP}}(q).$$

Some proof rules for procedures are given below.

$$\textsc{Call}\quad \frac{\{p\}\ k(\bar{x})\ \{q\} \in \Gamma}{\Gamma \vdash^{\mathsf{HLP}} \{p[\bar{E}/\bar{x}]\}\ y := k(\bar{E})\ \{q[\bar{E}/\bar{x}, y/\mathtt{ret}]\}}$$

$$\textsc{Prog}\quad \frac{\emptyset \vdash^{\mathsf{HLP}} \{p_1\}\ C_1\ \{q_1\}\ \ \cdots\ \ \emptyset \vdash^{\mathsf{HLP}} \{p_n\}\ C_n\ \{q_n\} \quad \{p_1\}\ k_1(\bar{x}_1)\ \{q_1\}, \ldots, \{p_n\}\ k_n(\bar{x}_n)\ \{q_n\} \vdash^{\mathsf{HLP}} \{p\}\ C\ \{q\}}{\vdash^{\mathsf{HLP}} \{p\}\ \mathtt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \mathtt{in}\ C\ \{q\}}$$

Modules Besides procedures, this dissertation also considers modules. Following O'Hearn et al. [2004], we consider a module to be "just a grouping of procedures that share some private state". The operational semantics for procedures already given in this section applies equally to modules, because the notion of "private state" does not exist in the program: only in the proof rules. The proof rules already given in this section have limited usefulness when applied to modules, because they do not enforce privacy of the module's internal state. We shall return to this issue in Chapter 4.

2.4 Abstract predicates

In Hoare logic, one often defines additional predicates in order to abbreviate assertions, or to clarify their meaning. To borrow an example from our later memory manager case study: consider a procedure `setbusy` that takes a pointer `p` and returns it having set its least significant bit. The following code implements this as a C macro:

```
#define setbusy(p) (struct store *)((int)(p)|1).
```

We might specify this procedure as follows:

$$\{true\}\ \mathtt{setbusy(p)}\ \{\mathtt{ret} = busy(\mathtt{p})\}\,. \tag{2.1}$$

The use of the '$busy$' predicate suggests how to understand the specification. It also provides a degree of abstraction. If one were later to change the implementation of `setbusy` to set, say, the *second least* significant bit, then existing proofs that use the specification above – but do not unfold the definition of $busy$ – should remain valid. For the current implementation, we define the predicate as follows:

$$busy(x) \stackrel{\text{def}}{=} x - \tfrac{1}{\mathtt{WORD}} \in \mathbb{N}. \tag{2.2}$$

(As explained on page 30, we treat pointers as being divided by `WORD`, the number of bytes in each word. This ensures that in a byte-addressed memory, if x is a pointer then $x + 1$ refers to the next *word*, not the next *byte*.)

This use of predicates to provide abstraction is formalised in an extension of Hoare logic called *abstract predicates*, due to Parkinson and Bierman [2005]. Judgements are augmented with a *predicate dictionary* Δ that contains a set of predicate definitions, of which (2.2) is an example. Any predicate that appears in the dictionary may be freely replaced with its definiens, and vice versa. Any predicate that does not appear in the dictionary is deemed abstract: it can be used in proofs, but its definition remains hidden. In the context of our `setbusy` example,

we arrange that the $busy$ predicate is in the dictionary only within the body of the `setbusy` procedure. This means that proofs outside of this scope are independent of the definition of $busy$, and hence that the definition can be changed without voiding those proofs.

To formalise abstract predicates, we first introduce a new construction to the syntax of assertions

$$\begin{aligned} p ::= & \ldots \\ & \mid \alpha(e, \ldots, e) \qquad \text{(abstract predicate)} \end{aligned}$$

where α is drawn from the set $\mathbb{A}$ of abstract predicate names, each associated with an arity. As in the previous section, we shall tacitly assume that when an abstract predicate is used, its list of arguments is of the correct length.

A *predicate environment* δ is a function of type:

$$\mathbb{A} \to (\mathsf{Val} \times \cdots \times \mathsf{Val}) \to \mathcal{P}(\mathsf{LState}).$$

The semantics of an assertion becomes parameterised by an predicate environment. This environment distributes in the expected way through the semantics already given; for instance, the semantics of conjunction becomes $[\![p \wedge q]\!](\delta) = [\![p]\!](\delta) \cap [\![q]\!](\delta)$. The semantics of an abstract predicate is as follows:

$$[\![\alpha(e_1, \ldots, e_n)]\!](\delta) = \delta(\alpha)([\![e_1]\!](\sigma, i), \ldots, [\![e_n]\!](\sigma, i)).$$

A predicate dictionary Δ is a finite partial function of type

$$\mathbb{A} \rightharpoonup_{\text{fin}} (\mathsf{LVar} \times \cdots \times \mathsf{LVar} \times \mathsf{Assertion})$$

that contains mappings such as $busy \mapsto (x, x - \frac{1}{\mathtt{WORD}} \in \mathbb{N})$. The semantics of a predicate dictionary is a set of predicate environments, each of which is a possible completion of the dictionary with definitions for all the missing predicate names; that is:

$$[\![\Delta]\!] = \{\delta \mid \forall (\alpha \mapsto (\bar{x}, p)) \in \Delta.\, \forall \bar{v}.\, \delta(\alpha)(\bar{v}) = [\![p[\bar{v}/\bar{x}]]\!](\delta)\}.$$

Judgements about the validity of assertions are now parameterised by a predicate dictionary; that is, we define

$$\Delta \models p \quad \stackrel{\text{def}}{=} \quad \forall \delta \in [\![\Delta]\!].\, ([\![p]\!](\delta) = \mathsf{LState}).$$

We now present an extension of Hoare logic with both procedures and abstract predicates. (Abstract predicates can be understood without procedures, but the abstraction they provide only becomes useful in the presence of procedures.)

Definition 2.5 (Meaning of judgements: Hoare logic with procedures and abstract predicates)**.** Let $safe_{\mathsf{HLA}}(q, \eta, \delta)$ be the largest set containing only those triples (C, σ, i) that:

- do not fault:
$$(C, \sigma) \not\rightarrow_\eta \mathbf{abort},$$
- satisfy the postcondition q if they are terminal:
$$C = \mathtt{skip} \Longrightarrow (\sigma, i) \in [\![q]\!](\delta),$$

- and continue to satisfy these properties after any execution step:

$$\forall C', \sigma'.\, (C, \sigma) \rightarrow_\eta (C', \sigma') \Longrightarrow (C', \sigma', i) \in \mathit{safe}_{\mathsf{HLA}}(q, \eta, \delta).$$

This is similar to Defn. 2.3, but with an additional parameter δ. We can then define:

$$\begin{aligned}
\models^{\mathsf{HLP}}_{\eta,\delta} \{p\}\ C\ \{q\} &\stackrel{\text{def}}{=} \forall (\sigma, i) \in [\![p]\!](\delta).\, (C, \sigma, i) \in \mathit{safe}_{\mathsf{HLP}}(q, \eta, \delta) \\
\eta\ \mathsf{implements}^{\mathsf{HL}}_{\delta}\ \Gamma &\stackrel{\text{def}}{=} \forall \{p\}\ k(\bar{x})\ \{q\} \in \Gamma.\, \exists C.\, \eta(k) = (\bar{x}, C) \wedge \models^{\mathsf{HLP}}_{\eta,\delta} \{p\}\ C\ \{q\} \\
\Delta; \Gamma \models^{\mathsf{HLP}} \{p\}\ C\ \{q\} &\stackrel{\text{def}}{=} \forall \delta \in [\![\Delta]\!].\, \forall \eta.\, \eta\ \mathsf{implements}^{\mathsf{HL}}_{\delta}\ \Gamma \Longrightarrow \models^{\mathsf{HLP}}_{\eta,\delta} \{p\}\ C\ \{q\}
\end{aligned}$$

Definition 2.6 (Meaning of judgements: Hoare logic with procedures and abstract predicates, for complete programs)**.** Let $\mathit{safe}_{\mathsf{HLA}}(q, \delta)$ be the largest set containing only those triples $(\mathit{prog}, \sigma, i)$ that:

- do not fault:

$$(\mathit{prog}, \sigma) \not\rightarrow \mathbf{abort},$$

- satisfy the postcondition q if they are terminal:

$$(\exists \eta.\, \mathit{prog} = \mathtt{procs}\ \eta\ \mathtt{in}\ \mathtt{skip}) \Longrightarrow (\sigma, i) \in [\![q]\!](\delta),$$

- and continue to satisfy these properties after any execution step:

$$\forall \mathit{prog}', \sigma'.\, (\mathit{prog}, \sigma) \rightarrow (\mathit{prog}', \sigma') \Longrightarrow (\mathit{prog}', \sigma', i) \in \mathit{safe}_{\mathsf{HLA}}(q, \delta).$$

We can then define:

$$\Delta \models^{\mathsf{HLA}} \{p\}\ \mathit{prog}\ \{q\} \stackrel{\text{def}}{=} \forall \delta \in [\![\Delta]\!].\, \forall (\sigma, i) \in [\![p]\!](\delta).\, (\mathit{prog}, \sigma, i) \in \mathit{safe}_{\mathsf{HLA}}(q, \delta).$$

Figure 2.4 presents some proof rules for abstract predicates. We write $\mathit{apn}(p)$ for the set of abstract predicate names appearing in the assertion p, and $\mathit{apn}(\Delta)$ for the set of abstract predicate names appearing in the definitions in the predicate dictionary Δ

The ABSTPREDI rule allows new predicate definitions to be introduced, and the ABSTPREDE rule allows the definitions of predicates that do not appear in the pre- or postcondition, or elsewhere in the predicate dictionary, to be eliminated. The PROG-ABST rule is for reasoning about procedures using abstract predicates. The departures from the PROG rule are the addition of the predicate dictionary Δ when verifying the procedure bodies $C_1, \ldots, C_n$, and the addition of the empty predicate dictionary $\emptyset$ when verifying the program's 'main' procedure C. (In fact, the PROG-ABST rule is derivable from the PROG, ABSTPREDI and ABSTPREDE rules.) The intention is that the preconditions $(p_1, \ldots, p_n)$ and postconditions $(q_1, \ldots, q_n)$ of the procedures may involve some abstract predicates, which are defined in Δ. By making different predicate dictionaries available to the different parts of the program in this way, we ensure that the predicate definitions are available only when reasoning inside the procedure bodies.

$$\textsc{Abst-Conseq}\ \frac{\Delta;\Gamma \vdash^{\mathsf{HLP}} \{p'\}\ C\ \{q'\} \qquad \Delta \models p \Rightarrow p' \qquad \Delta \models q' \Rightarrow q}{\Delta;\Gamma \vdash^{\mathsf{HLP}} \{p\}\ C\ \{q\}}$$

$$\textsc{AbstPredE}\ \frac{\Delta \uplus \Delta' \vdash \{p\}\ prog\ \{q\} \qquad \mathrm{dom}(\Delta') \not\cap (apn(p) \cup apn(q) \cup apn(\Delta))}{\Delta \vdash \{p\}\ prog\ \{q\}}$$

$$\textsc{AbstPredI}\ \frac{\Delta;\Gamma \vdash^{\mathsf{HLP}} \{p\}\ C\ \{q\}}{\Delta \uplus \Delta';\Gamma \vdash^{\mathsf{HLP}} \{p\}\ C\ \{q\}}$$

$$\textsc{Prog-Abst}\ \frac{\Delta;\emptyset \vdash^{\mathsf{HLP}} \{p_1\}\ C_1\ \{q_1\}\ \ \cdots\ \ \Delta;\emptyset \vdash^{\mathsf{HLP}} \{p_n\}\ C_n\ \{q_n\} \qquad \emptyset;(\{p_1\}\ k_1(\bar{x}_1)\ \{q_1\},\ldots,\{p_n\}\ k_n(\bar{x}_n)\ \{q_n\}) \vdash^{\mathsf{HLP}} \{p\}\ C\ \{q\}}{\Delta \vdash^{\mathsf{HLP}} \{p\}\ \mathtt{procs}\ k_1(\bar{x}_1) \stackrel{\mathrm{def}}{=} C_1,\ldots,k_n(\bar{x}_n) \stackrel{\mathrm{def}}{=} C_n\ \mathtt{in}\ C\ \{q\}}$$

Figure 2.4: Proof rules for abstract predicates

2.5 Separation logic

Separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] is an extension of Hoare logic that is able to reason effectively about programs, such as those written in a C-like language, that compute with mutable, dynamically-allocated memory.

In the context of separation logic, a program state $\sigma \in \mathsf{PState}$ is interpreted as a pair (s, h), comprising a *store* s and a *heap* h. (We shall later also use H to range over heaps.) The store is the new name for what was previously the entire state: it is a finite partial function that maps program variables to values. The heap describes the contents of dynamically-allocated memory locations. It is a finite partial function that maps addresses (typically positive integers) to values, and is only defined for those locations that are marked as allocated.

$$\begin{aligned} \mathsf{Store} &\stackrel{\mathrm{def}}{=} \mathsf{PVar} \to \mathsf{Val} \\ \mathsf{Heap} &\stackrel{\mathrm{def}}{=} \mathbb{N}^+ \rightharpoonup_{\mathrm{fin}} \mathsf{Val} \\ \mathsf{PState} &\stackrel{\mathrm{def}}{=} \mathsf{Store} \times \mathsf{Heap} \end{aligned}$$

We lift the $\uplus$ operator to act on heaps, such that $h \uplus h'$ is defined, when h and h' have disjoint domains, to contain all the mappings that are in h or h'. Let us write $\emptyset$ for the empty heap; that is, the completely undefined partial function. Let us write $h_1 \sqsubseteq h_2$, and say that h_1 is a *subheap* of h_2, when h_1's domain is a subset of h_2's domain and they contain equal values wherever they are both defined.

We now extend our programming language with some new basic commands for interacting with the heap.

$$\begin{aligned} c ::= \ & \ldots \\ & \mid \mathtt{[}E_1\mathtt{]} := E_2 && \text{(heap update)} \\ & \mid x := \mathtt{[}E\mathtt{]} && \text{(heap lookup)} \\ & \mid x := \mathtt{alloc}() && \text{(allocation)} \\ & \mid \mathtt{dispose}(E) && \text{(disposal)} \end{aligned}$$

The operational semantics of these basic commands is as follows.

- If E_1 and E_2 are program expressions that evaluate to v_1 and v_2 respectively, and v_1 is an allocated heap location, then $\texttt{[}E_1\texttt{]} := E_2$ replaces the contents of heap location v_1 with the new value v_2.

$$[\![\texttt{[}E_1\texttt{]} := E_2]\!](s,h) \;=\; \begin{cases} \{(\texttt{skip}, (s, h[v_1 \mapsto v_2]))\} & \text{if } v_1 \in \mathrm{dom}(h) \\ \textbf{abort} & \text{otherwise} \end{cases}$$
$$\text{where } v_1 = [\![E_1]\!](s) \text{ and } v_2 = [\![E_2]\!](s)$$

- If x is a program variable, E is a program expression that evaluates to v, and v is an allocated heap location, then $x := \texttt{[}E\texttt{]}$ loads the contents of heap location v into x.

$$[\![x := \texttt{[}E\texttt{]}]\!](s,h) \;=\; \begin{cases} \{(\texttt{skip}, (s[x \mapsto h(v)], h))\} & \text{if } v \in \mathrm{dom}(h) \\ \textbf{abort} & \text{otherwise} \end{cases}$$
$$\text{where } v = [\![E]\!](s)$$

- If x is a program variable, then $x := \texttt{alloc}()$ chooses an unallocated heap location, flags it as allocated, zeroes its contents, and stores its address in x.

$$[\![x := \texttt{alloc}()]\!](s,h) \;=\; \{(\texttt{skip}, (s[x \mapsto v], h[v \mapsto 0])) \mid v \notin \mathrm{dom}(h)\}$$

- If E is a program expression that evaluates to v, and v is an allocated heap location, then $\texttt{dispose}(E)$ marks it as unallocated.

$$[\![\texttt{dispose}(E)]\!](s,h) \;=\; \begin{cases} \{(\texttt{skip}, (s, h \setminus \{v\}))\} & \text{if } v \in \mathrm{dom}(h) \\ \textbf{abort} & \text{otherwise} \end{cases}$$
$$\text{where } v = [\![E]\!](s)$$

Separation logic introduces several new constructions for building assertions.

$$\begin{array}{lll} p ::= \ldots & & \\ \quad \mid p * p & \text{(separating conjunction)} & \\ \quad \mid p \mathrel{-\!\!*} p & \text{(separating implication)} & \\ \quad \mid emp & \text{(empty heap)} & \\ \quad \mid e \mapsto e & \text{(singleton heap)} & \\ \quad \mid \circledast_{i=0}^{n} p_i & \text{(iterated separating conjunction)} & \end{array} \tag{2.3}$$

The semantics of these assertions is given below.

- A state satisfies $p * q$ if it can be split into two smaller states, one satisfying p and the other satisfying q.

$$[\![p * q]\!] \;=\; \{(s, h_1 \uplus h_2, i) \mid (s, h_1, i) \in [\![p]\!] \wedge (s, h_2, i) \in [\![q]\!]\}$$

  The basic distinction between $\wedge$ and $*$ is that the assertion $p \wedge q$ holds when p and q hold in the *same* heap, whereas $p * q$ holds when p and q hold in *non-overlapping* heaps.

  Remark 2.7. These two versions of 'and' can be identified in ordinary English. For instance, the dedication of this thesis is: *To my wife and best friend*. Depending on whether the 'and' is interpreted as $\wedge$ or $*$, the dedication is either to one person or to two.

- A state satisfies emp if its heap is empty.

$$[\![emp]\!] \;=\; \{(s,h,i) \mid \mathrm{dom}(h) = \emptyset\}$$

  Note that emp is the unit of $*$, just as $true$ is the unit of $\wedge$. Extending this analogy, separation logic also provides an operator corresponding to $\Rightarrow$, which is called 'separating implication' and written $-\!\!*$.

- A state satisfies $p -\!\!* q$ if whenever it is extended with a separate state that satisfies p, the result satisfies q.

$$\begin{array}{rcl}[\![p -\!\!* q]\!] & = & \{(s,h_1,i) \mid \forall h_2.\, \mathrm{dom}(h_1) \not\asymp \mathrm{dom}(h_2) \wedge (s,h_2,i) \in [\![p]\!] \\ & & \qquad \Rightarrow (s, h_1 \uplus h_2, i) \in [\![q]\!]\}\end{array}$$

- A state satisfies $e_1 \mapsto e_2$ if its heap comprises a single cell at address e_1 with contents e_2.

$$\begin{array}{rcl}[\![e_1 \mapsto e_2]\!] & = & \{(s,h,i) \mid \exists v_1, v_2.\, [\![e_1]\!](s,i) = v_1 \wedge [\![e_2]\!](s,i) = v_2 \wedge v_1 \in \mathbb{N}^+ \\ & & \qquad \wedge\, \mathrm{dom}(h) = \{v_1\} \wedge h(v_1) = v_2\}\end{array}$$

  The definition above differs from traditional presentations of separation logic, in that we treat memory as byte-addressed rather than word-addressed. This is appropriate for our memory manager case study, which exploits the redundancy of the least significant bits of a word-aligned pointer in a byte-addressed memory. Pointers are fractions in the following set

$$\mathtt{ptr} \;\stackrel{\mathrm{def}}{=}\; \{x \mid x \times \mathtt{WORD} \in \mathbb{Z}\}$$

  where `WORD` is the number of bytes in a word. The semantics of the single-cell assertion requires the address to be word-aligned, that is, to evaluate to a positive natural number. Note that if x is a pointer, then '$x + 1$' denotes the next *word* rather than the next *byte*. We can access the lower bits of a pointer by adding or subtracting fractions. A downside of this treatment is that having effectively divided all pointers by `WORD`, we must be very careful when comparing pointers with non-pointers.

  It is convenient to introduce a couple of shorthands here. Let us write $e \mapsto \_$ as shorthand for $\exists x.\, e \mapsto x$, and let $e \mapsto e_1 \;\ldots\; e_n$ abbreviate $(e \mapsto e_1) * \cdots * ((e+n-1) \mapsto e_n)$.

- Finally, iterated separating conjunction is simply the n-ary form of binary separating conjunction.

$$[\![\circledast_{i=0}^{n} p_i]\!] \;=\; \{(s, h_0 \uplus \cdots \uplus h_n, i) \mid (s,h_0,i) \in [\![p_0]\!] \wedge \ldots \wedge (s,h_n,i) \in [\![p_n]\!]\}$$

The semantics of separation logic judgements, presented below, is sensitive to the fact that although assertions may refer only to a small set of heap locations, the program may actually be running on a larger heap.

Definition 2.8 (Meaning of judgements: separation logic)**.** Let $safe_{\mathsf{SL}}(q)$ be the largest set containing only those quadruples (C, s, h, i) that:

- do not fault (even in the presence of extra heap locations h_o):

$$\forall h_o, h_1.\, h \uplus h_o = h_1 \Longrightarrow (C, (s, h_1)) \not\rightarrow \mathbf{abort},$$

$$\textsc{HeapUpdate}\ \frac{}{\vdash \{e_1 \mapsto \_\}\ [e_1] := e_2\ \{e_1 \mapsto e_2\}} \qquad \textsc{HeapLookup}\ \frac{}{\vdash \{e = Y \wedge Y \mapsto X\}\ x := [e]\ \{Y \mapsto X \wedge x = X\}}$$

$$\textsc{Allocation}\ \frac{}{\vdash \{emp\}\ x := \mathtt{alloc}()\ \{x \mapsto \_\}} \qquad \textsc{Disposal}\ \frac{}{\vdash \{e \mapsto \_\}\ \mathtt{dispose}(e)\ \{emp\}}$$

$$\textsc{Frame}\ \frac{\vdash \{p\}\ C\ \{q\} \qquad wr(C) \not\frown rd(r)}{\vdash \{p * r\}\ C\ \{q * r\}}$$

Figure 2.5: Proof rules for separation logic

- satisfy the postcondition q if they are terminal:

$$C = \mathtt{skip} \implies (s, h, i) \in [\![q]\!],$$

- and continue to satisfy these properties after any execution step (noting that this execution step may occur in the presence of extra heap locations h_o that are unaffected by the step):

$$\forall h_o, h_1, C', s', h_1'.\, h \uplus h_o = h_1 \wedge (C, (s, h_1)) \to (C', (s', h_1')) \\ \implies (\exists h'.\, h_1' = h' \uplus h_o \wedge (C', s', h', i) \in safe_{\mathsf{SL}}(q)).$$

We can then define:

$$\models_{\mathsf{SL}} \{p\}\ C\ \{q\} \quad \stackrel{\text{def}}{=} \quad \forall (s, h, i) \in [\![p]\!].\, (C, s, h, i) \in safe_{\mathsf{SL}}(q).$$

Figure 2.5 presents several proof rules of separation logic. Note that all the rules from Hoare logic remain valid. The FRAME rule employs the following definition.

Definition 2.9 (Reading and writing program variables)**.** The rd and wr functions respectively extract the sets of program variables read and written. They can be applied to a variety of objects, such as assertions (which only read) and commands. We write $X \# Y$ when both $rd(X) \not\frown wr(Y)$ and $rd(Y) \not\frown wr(X)$.

Example 2.10 (Three heap updates)**.** If `x`, `y` and `z` are distinct heap locations, initially containing zeroes, then the program

$$[\mathtt{x}] := 1\ ;\ [\mathtt{y}] := 1\ ;\ [\mathtt{z}] := 1$$

can be verified in separation logic like so:

$$\dfrac{\dfrac{\dfrac{}{\{\mathtt{x} \mapsto 0\}\,[\mathtt{x}] := 1\,\{\mathtt{x} \mapsto 1\}}\,\mathrm{U}}{\{p_{000}\}\,[\mathtt{x}] := 1\,\{p_{100}\}}\,\mathrm{F} \qquad \dfrac{\dfrac{\dfrac{}{\{\mathtt{y} \mapsto 0\}\,[\mathtt{y}] := 1\,\{\mathtt{y} \mapsto 1\}}\,\mathrm{U}}{\{p_{100}\}\,[\mathtt{y}] := 1\,\{p_{110}\}}\,\mathrm{F} \qquad \dfrac{\dfrac{}{\{\mathtt{z} \mapsto 0\}\,[\mathtt{z}] := 1\,\{\mathtt{z} \mapsto 1\}}\,\mathrm{U}}{\{p_{110}\}\,[\mathtt{z}] := 1\,\{p_{111}\}}\,\mathrm{F}}{\{p_{100}\}\,[\mathtt{y}] := 1\ ;\ [\mathtt{z}] := 1\,\{p_{111}\}}\,\mathrm{S}}{\{p_{000}\}\,[\mathtt{x}] := 1\ ;\ [\mathtt{y}] := 1\ ;\ [\mathtt{z}] := 1\,\{p_{111}\}}\,\mathrm{S}$$

where S, F and U stand for applications of the SEQ, FRAME and HEAPUPDATE rules respectively, and p_{ijk} abbreviates $\mathtt{x} \mapsto i * \mathtt{y} \mapsto j * \mathtt{z} \mapsto k$. A less redundant representation of this proof is the following *proof outline*, which intersperses the program's instructions with 'enough' assertions to allow the reader to reconstruct the derivation tree.

$\{\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$
`[x]:=1;`
$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 0 * \mathtt{z} \mapsto 0\}$
`[y]:=1;`
$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 0\}$
`[z]:=1;`
$\{\mathtt{x} \mapsto 1 * \mathtt{y} \mapsto 1 * \mathtt{z} \mapsto 1\}$

We shall investigate an even less redundant representation of this proof in Chapter 5.

The key point of the example above is that, at the leaves of the tree, we reason about each heap update as if it were operating on a heap containing *only* the location being updated. (For this reason, the HEAPUPDATE, HEAPLOOKUP, ALLOCATION and DISPOSAL rules are often called the *small axioms*.) Then we use the FRAME rule to embed this into a larger heap containing the other two locations as well.

2.5.1 Partition diagrams

We shall frequently wish to describe a series of consecutive heap cells in this dissertation. This can be done using the iterated separating conjunction, but here we introduce a new notation, inspired by the *partition diagrams* invented by Reynolds [1979]. First, let us write the single-cell assertion as $\left|\begin{smallmatrix} e \\ e' \end{smallmatrix}\right|$ instead of $e \mapsto e'$. Although this new notation sacrifices linearity, it extends better to ranges of cells, which we write as $\left|\overset{e_1}{\text{———}}\right|^{e_2}$. These multiple-cell assertions have the following meaning.

$$\left[\!\left[\left|\overset{e_1}{\text{———}}\right|^{e_2}\right]\!\right] \quad = \quad e_1 \leq e_2 \wedge \circledast_{i=0}^{e_2-e_1-1} (e_1 + i \mapsto \_)$$

The imposition that e_1 does not exceed e_2 makes our diagrams *regular*, according to Reynolds' terminology. The default diagram has an inclusive lower bound and an exclusive upper bound, but alternatives can be obtained by moving the variables across the vertical dividers. The assertions

$$\left|\overset{e_1}{\text{———}}\right|^{e_2} \quad \text{and} \quad \left|\overset{e_1}{\text{———}}\right|^{e_2-1} \quad \text{and} \quad {}^{e_1-1}\left|\overset{}{\text{———}}\right|^{e_2-1} \quad \text{and} \quad {}^{e_1-1}\left|\overset{}{\text{———}}\right|^{e_2}$$

are all equivalent. Ranges may be concatenated; that is,

$$\left|\overset{e_1}{\text{———}}\right|^{e_2} * \left|\overset{e_2}{\text{———}}\right|^{e_3} \quad \text{and} \quad \left|\overset{e_1}{\text{———}}\overset{e_2}{\text{———}}\right|^{e_3}$$

are equivalent, and $\left|\overset{e_1}{\text{———}}\right|^{e_3}$ can be deduced from either. We can write $\left|\overset{e}{\text{———}}\overset{e}{\text{———}}\right|$ as $\left|\begin{smallmatrix} e \\ | \end{smallmatrix}\right|$.

2.5.2 Fractional permissions

The assertion '$\mathtt{x} \mapsto 3$' denotes a heap containing just one cell, but there are times (mostly when dealing with concurrency) when it is conceptually helpful to divide this assertion even further.

One can think of '$\mathtt{x} \mapsto 3$' not just as a statement that location x holds value 3, but as a token that confers upon the holder permission to update that value to, say, 4:

$$\left\{\mathtt{x} \mapsto 3\right\}\ \texttt{[x]:=4}\ \left\{\mathtt{x} \mapsto 4\right\}.$$

The FRAME rule is sound when applied to this specification precisely because the framed assertion will be prohibited by the $*$-operator from referring to the contents of x, and hence cannot be invalidated by the update.

Bornat et al. [2005] describe a variant of separation logic that allows assertions of the form $\pi(e \mapsto e')$, where $\pi \in \mathsf{Frac}$ and

$$\mathsf{Frac} \stackrel{\text{def}}{=} \{\pi \in \mathbb{Q} \mid 0 < \pi \leq 1\}.$$

The assertion $\pi(e \mapsto e')$ confers permission to write to location e when $\pi = 1$, and read-only access when $\pi < 1$, as manifested in the following updated proof rules (in which we abbreviate $1(e \mapsto e')$ as $e \mapsto e'$):

HEAPUPDATE-PERM

$$\frac{}{\vdash \left\{e_1 \mapsto \_\right\}\ \texttt{[}e_1\texttt{]} := e_2\ \left\{e_1 \mapsto e_2\right\}}$$

HEAPLOOKUP-PERM

$$\frac{}{\vdash \left\{e = Y \wedge \pi(Y \mapsto X)\right\}\ x := \texttt{[}e\texttt{]}\ \left\{\pi(Y \mapsto X) \wedge x = X\right\}}$$

Our notion of program state must be updated to use 'fractional heaps', which associate each address with both a value and a fraction:

$$\begin{aligned} \mathsf{FractionalHeap} &\stackrel{\text{def}}{=} \mathbb{N}^+ \rightharpoonup_{\text{fin}} (\mathsf{Val} \times \mathsf{Frac}) \\ \mathsf{PState} &\stackrel{\text{def}}{=} \mathsf{Store} \times \mathsf{FractionalHeap} \end{aligned}$$

Fractions can be combined by a partial $+$-operator that is undefined when the result would exceed 1. This operator interacts with the $*$-operator as follows:

$$(\pi_1 + \pi_2)(e \mapsto e_1) \wedge e_1 = e_2 \quad = \quad \pi_1(e \mapsto e_1) * \pi_2(e \mapsto e_2).$$

The $+$-operator on fractions can be lifted to fractional heaps, and thence into the semantics of the $*$-operator, like so:

$$h_1 + h_2 = \{(l \mapsto (v, \pi_1 + \pi_2)) \mid (l \mapsto (v, \pi_1)) \in h_1 \wedge (l \mapsto (v, \pi_2)) \in h_2\} \tag{2.4}$$

$$[\![p * q]\!] = \{(s, h_1 + h_2, i) \mid (s, h_1, i) \in [\![p]\!] \wedge (s, h_2, i) \in [\![q]\!]\}. \tag{2.5}$$

2.5.3 Variables as resource

One dissatisfaction with separation logic, as presented thus far, is its rather unsophisticated treatment of program variables. Consider the FRAME rule. If the triple $\left\{p\right\}\ C\ \left\{q\right\}$ can be proved valid, then any heap location accessed by C is surely specified by the precondition p (or else allocated during the execution of C). Then we can also prove the triple $\left\{p * r\right\}\ C\ \left\{q * r\right\}$ valid. The assertion r describes only heap locations that are not described by p, and are hence not accessed by C, so r will remain valid in the postcondition. This reasoning considers only the heap, however. For soundness of the FRAME rule, we require also the side-condition that r does not mention any program variables that C might modify.

Variables-as-resource is a variation of separation logic in which the treatment of program variables more closely resembles the treatment of the heap. The original version, which we follow in this dissertation, is due to Bornat et al. [2006].

Remark 2.11. A revised version, introduced by Parkinson et al. [2006], defines the elements of Store to be *partial* functions. A partial store is sensible for modelling programming languages whose variables do not all have global scope, but it does have oddities such as $\mathtt{x} = \mathtt{x}$ being different from $true$ (it is false if x is not defined in the store). A total store enables a simpler logic without such oddities.

In variables-as-resource, the $*$-operator separates not only the heap, but program variables as well. As a result, the FRAME rule can be rephrased without the side-condition.

We do not build on the 'fractional heaps' model presented in the previous subsection, though we shall inherit the definition of Frac. We update the notion of the logical state to include a *permissions mask*, ranged over by Π, that records which fraction of each variable is owned. Note that this permissions mask exists only in the proof, and is not part of the program state, which remains unchanged.

$$\begin{aligned} \mathsf{PermMask} &\stackrel{\text{def}}{=} \mathsf{PVar} \rightharpoonup_{\text{fin}} \mathsf{Frac} \\ \mathsf{PState} &\stackrel{\text{def}}{=} \mathsf{Store} \times \mathsf{Heap} \\ \mathsf{LState} &\stackrel{\text{def}}{=} \mathsf{PState} \times \mathsf{PermMask} \times (\mathsf{LVar} \to \mathsf{Val}) \end{aligned}$$

A new form of assertion is required.

$$\begin{aligned} p ::= {} & \ldots \\ & \mid \mathit{Own}_\pi(x) \qquad \text{(ownership of a program variable)} \end{aligned}$$

The assertion $\mathit{Own}_\pi(x)$ denotes ownership of a fraction $\pi \in \mathsf{Frac}$ of the program variable x. When $\pi = 1$, this assertion confers permission to write to x, and when $\pi < 1$, the variable may be read. We shall write

$$\pi_1 x_1, \ldots, \pi_n x_n \Vdash p$$

as shorthand for

$$\mathit{Own}_{\pi_1}(x_1) * \cdots * \mathit{Own}_{\pi_n}(x_n) * p.$$

The $\Vdash$ connective has the lowest precedence of all. Let O range over lists of variables decorated with fractions such as the list $\pi_1 x_1, \ldots, \pi_n x_n$ above, and let us write x in place of $1x$. We define an assertion p to be *well-supported* when it only depends on program variables of which it claims some ownership; that is, when

$$(s, h, \Pi, i) \in [\![p]\!] \quad \text{implies} \quad \forall s'.\, (\forall x \in \mathrm{dom}(\Pi).\, s'(x) = s(x)) \Longrightarrow (s', h, \Pi, i) \in [\![p]\!].$$

Remark 2.12. Later, when the rely-guarantee method is introduced, we shall encounter the notion of *stability*. A stable assertion cannot be invalidated by interference from other threads, in much the same way as a well-supported assertion cannot be invalidated by changes to program variables of which it has no ownership.

In the proof rules, which are given in Fig. 2.6, we shall implicitly require that every pre- and postcondition is well-supported. Well-supportedness is easily attained when writing assertions in the form $O \Vdash p$ by ensuring that all of the program variables in p appear in O.

Note that substitution does not affect ownership predicates; that is,

$$\mathit{Own}_\pi(x)[E/x] \quad = \quad \mathit{Own}_\pi(x).$$

The semantics of ownership assertions is as follows.

$$[\![\mathit{Own}_\pi(x)]\!] \quad = \quad \{(s, h, \Pi, i) \mid \mathrm{dom}(\Pi) = \{x\} \wedge \Pi(x) = \pi \wedge \mathrm{dom}(h) = \emptyset\}$$

$$\textsc{Assume-VaR}\quad \frac{}{\vdash_{\mathsf{VaR}} \{O \Vdash p\}\ \texttt{assume}\, B\ \{O \Vdash p \wedge B\}} \qquad \textsc{Assert-VaR}\quad \frac{}{\vdash_{\mathsf{VaR}} \{O \Vdash p \wedge B\}\ \texttt{assert}\, B\ \{O \Vdash p\}}$$

$$\textsc{Assign-VaR}\quad \frac{}{\vdash_{\mathsf{VaR}} \{x, O \Vdash p[E/x]\}\ x := E\ \{x, O \Vdash p\}} \qquad \textsc{Frame-VaR}\quad \frac{\vdash_{\mathsf{VaR}} \{p\}\ C\ \{q\}}{\vdash_{\mathsf{VaR}} \{p * r\}\ C\ \{q * r\}}$$

Figure 2.6: Proof rules for variables-as-resource

The semantics of the other assertion constructors should be updated by adding the empty permissions mask. For instance, 'emp' describes both an empty heap and an empty permissions mask.

$$[\![emp]\!] \quad = \quad \{(s, h, \Pi, i) \mid \mathrm{dom}(\Pi) = \mathrm{dom}(h) = \emptyset\}$$

As in the previous subsection, we lift the partial $+$-operator from Frac to permissions masks, and thence into the semantics of the $*$-operator, like so:

$$\begin{array}{rcl} \Pi_1 + \Pi_2 & = & \{(x \mapsto \pi_1 + \pi_2) \mid (x \mapsto \pi_1) \in \Pi_1 \wedge (x \mapsto \pi_2) \in \Pi_2\} \\ [\![p * q]\!] & = & \{(s, h_1 \uplus h_2, \Pi_1 + \Pi_2, i) \mid (s, h_1, \Pi_1, i) \in [\![p]\!] \wedge (s, h_2, \Pi_2, i) \in [\![q]\!]\}. \end{array}$$

Definition 2.13 (Meaning of judgements: variables-as-resource)**.** Let $safe_{\mathsf{VaR}}(q)$ be the largest set containing only those quadruples (C, s, h, Π, i) that:

- do not fault (even in the presence of extra heap locations h_o):

$$\forall h_o, h_1.\, h \uplus h_o = h_1 \implies (C, (s, h_1)) \not\rightarrow \mathbf{abort},$$

- satisfy the postcondition q if they are terminal:

$$C = \texttt{skip} \implies (s, h, \Pi, i) \in [\![q]\!],$$

- and continue to satisfy these properties after any execution step (noting that this execution step may occur in the presence of extra heap locations h_o that are unaffected by the step):

$$\begin{array}{l} \forall h_o, h_1, C', s', h_1'.\, h \uplus h_o = h_1 \wedge (C, (s, h_1)) \rightarrow (C', (s', h_1')) \\ \qquad\qquad \implies (\exists h'.\, h_1' = h' \uplus h_o \wedge (C', s', h', \Pi, i) \in safe_{\mathsf{SL}}(q)). \end{array}$$

We can then define:

$$\models_{\mathsf{VaR}} \{p\}\ C\ \{q\} \quad \stackrel{\text{def}}{=} \quad \forall (s, h, \Pi, i) \in [\![p]\!].\, (C, s, h, \Pi, i) \in safe_{\mathsf{VaR}}(q).$$

2.6 Reasoning about concurrency

We expand our attention to concurrent programming languages, and look at an early extension of Hoare logic for handling such languages, called the Owicki-Gries method.

Let us begin by adding to our programming language the following constructions for parallel composition and atomic blocks.

$$\begin{aligned} C ::= {} & \ldots \\ & \mid \texttt{atomic}\, C \\ & \mid C \parallel C \end{aligned}$$

The $\texttt{atomic}\, C$ command executes C atomically; that is, in a single uninterruptible step of execution. The $C_1 \parallel C_2$ command executes C_1 and C_2 in parallel by forming an arbitrary interleaving of their individual execution steps. Just as we used assumptions to encode if-statements and while-loops in Sect. 2.1, we can encode conditional atomic blocks – which only execute when a condition B is met – like so:

$$\texttt{when}\ B\ \texttt{atomic}\ C \stackrel{\text{def}}{=} \texttt{atomic}(\texttt{assume}\, B\ ;\ C).$$

The operational semantics of parallel composition and atomic blocks are as follows; note that we write $\rightarrow^*$ to denote a finite sequence of zero or more steps of computation.

$$\frac{(C,\sigma) \rightarrow^* (\texttt{skip}, \sigma')}{(\texttt{atomic}\, C, \sigma) \rightarrow (\texttt{skip}, \sigma')} \qquad \frac{(C,\sigma) \rightarrow^* \mathbf{abort}}{(\texttt{atomic}\, C, \sigma) \rightarrow \mathbf{abort}}$$

$$\frac{}{(\texttt{skip} \parallel \texttt{skip}, \sigma) \rightarrow (\texttt{skip}, \sigma)} \qquad \frac{(C_1,\sigma) \rightarrow (C_1', \sigma')}{(C_1 \parallel C_2, \sigma) \rightarrow (C_1' \parallel C_2, \sigma')} \qquad \frac{(C_1,\sigma) \rightarrow \mathbf{abort}}{(C_1 \parallel C_2, \sigma) \rightarrow \mathbf{abort}}$$

$$\frac{(C_2,\sigma) \rightarrow (C_2', \sigma')}{(C_1 \parallel C_2, \sigma) \rightarrow (C_1 \parallel C_2', \sigma')} \qquad \frac{(C_2,\sigma) \rightarrow \mathbf{abort}}{(C_1 \parallel C_2, \sigma) \rightarrow \mathbf{abort}}$$

Reasoning about concurrent programs is hard because commands from different threads are interleaved non-deterministically. With many threads and many commands per thread, reasoning easily succumbs to a combinatorial explosion. An early extension of Hoare Logic to concurrency, developed by Owicki and Gries [1976], proposed the following rule for reasoning about parallel composition.

OG-PAR

$$\frac{\vdash \{p_1\}\ C_1\ \{q_1\} \qquad \vdash \{p_2\}\ C_2\ \{q_2\} \qquad \text{the proofs of } C_1 \text{ and } C_2 \text{ are interference-free}}{\vdash \{p_1 \wedge p_2\}\ C_1 \parallel C_2\ \{q_1 \wedge q_2\}}$$

The first two antecedents require the two threads C_1 and C_2 to be proved in isolation, and the third requires that the two proofs do not 'interfere', which is roughly to say that none of the assertions in one thread's proof can be invalidated by executing a command in the other thread. The problem that Jones [1983] identifies with this rule is that one may spend much effort verifying C_1 and C_2, only to discover that the proofs need repeating, with different assertions, because the third condition on the OG-PAR rule fails upon assembling the complete program. The technical problem is that the Owicki-Gries method is not *compositional*. A proof system is

$$\text{CSL-BASIC}\quad \frac{\vdash \{p * J\}\, c\, \{q * J\}}{\vdash c\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J)} \qquad \text{CSL-ATOMIC}\quad \frac{\vdash \{p * J\}\, C\, \{q * J\}}{\vdash \mathtt{atomic}\, C\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J)}$$

$$\text{CSL-ATOMICCOND}\quad \frac{\vdash \{p * (J \wedge B)\}\, C\, \{q * J\}}{\vdash \mathtt{when}\ B\ \mathtt{atomic}\ C\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J)} \qquad \text{CSL-HIDE}\quad \frac{\vdash C\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J * J')}{\vdash C\ \mathbf{sat}_{\mathsf{CSL}}\ (p * J', q * J', J)}$$

$$\text{CSL-PAR}\quad \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{CSL}}\ (p_1, q_1, J) \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{CSL}}\ (p_2, q_2, J)}{\vdash C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{CSL}}\ (p_1 * p_2, q_1 * q_2, J)}$$

Figure 2.7: Proof rules for concurrent separation logic

compositional when judgements about the behaviour of a command can be made purely based on the judgements about its immediate subcommands, without additional knowledge of the interior construction of those subcommands. Jones argues that compositionality "would appear to be an essential requirement for a method to be useful for large problems."

The Owicki-Gries rule can be made compositional in a couple of ways, which we now discuss. The first is called *concurrent separation logic* and the second is the *rely-guarantee* method.

2.7 Concurrent separation logic

The following rule exploits separation logic to satisfy the non-interference condition in the OG-PAR rule.

$$\text{DISJOINTPAR}\quad \frac{\vdash \{p_1\}\ C_1\ \{q_1\} \qquad \vdash \{p_2\}\ C_2\ \{q_2\}}{\vdash \{p_1 * p_2\}\ C_1 \parallel C_2\ \{q_1 * q_2\}}$$

If the resources specified by p_1 and p_2 can be combined by the $*$-operator, then they must be disjoint. Hence the threads C_1 and C_2 that operate on those resources are isolated from each other, and their respective postconditions will be satisfied when they terminate. The rule is compositional, but it is not very powerful because it cannot handle programs whose threads communicate.

Concurrent separation logic, due to O'Hearn [2004] and Brookes [2004], allows some resources to be shared among multiple threads. It uses judgements of the form

$$\models C\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J)$$

where the *resource invariant* $J \in \mathsf{Assertion}$ describes the part of the state that is shared, while p and q describe only the local part. (The original presentation uses multiple resource invariants; this simpler version was first used by Parkinson et al. [2007].) Upon beginning execution of an atomic block, the shared state becomes available (and can be assumed to satisfy J), and upon finishing, the shared state is relinquished (and must satisfy J again).

Some proof rules from concurrent separation logic are given in Fig. 2.7. The CSL-BASIC and CSL-ATOMIC rules require a basic command or an atomic block to be verified with J appended to its pre- and postcondition. This allows the command to access the shared state,

which is sensible because basic commands and atomic blocks execute without interruption. The CSL-HIDE rule captures the fact that to prove a judgement, it suffices to prove a similar judgement in which some of the local state has been rebranded as shared. (To reflect this reading of the rule, Vafeiadis [2011] names it CSL-SHARE.) The CSL-PAR rule is the same as the DISJOINTPAR rule from ordinary separation logic, but with J distributed throughout its antecedents and consequent. In fact, all of the rules of separation logic can be extended to concurrent separation logic by adding the resource invariant in this way, except CONJ, which requires J to be *precise*.

Definition 2.14. An assertion p is precise if for every logical state (s, h, i), there exists at most one subheap $h' \sqsubseteq h$ for which $(s, h', i) \in [\![p]\!]$.

Definition 2.15 (Meaning of judgements: concurrent separation logic)**.** Let $safe_{\mathsf{CSL}}(q, J)$ be the largest set containing only those quadruples (C, s, h, i) that:

- do not fault (even in the presence of a shared heap $H \in \mathsf{Heap}$ satisfying J, and extra heap locations h_o):

$$\forall H, h_o, h_1.\, h \uplus H \uplus h_o = h_1 \wedge (s, H, i) \in [\![J]\!] \implies (C, (s, h_1)) \not\to \mathbf{abort},$$

- satisfy the postcondition q if they are terminal:

$$C = \mathtt{skip} \implies (s, h, i) \in [\![q]\!],$$

- and continue to satisfy these properties after any execution step (noting that this execution step may transform the shared heap H into H' providing J is preserved, and may occur in the presence of extra heap locations h_o that are unaffected by the step):

$$\begin{array}{l}\forall H, h_o, h_1, c', s', h_1'.\\ \quad h \uplus H \uplus h_o = h_1 \wedge (s, H, i) \in [\![J]\!] \wedge (C, (s, h_1)) \to (C', (s', h_1'))\\ \quad\implies (\exists h', H'.\, h_1' = h' \uplus H' \uplus h_o \wedge (s', H', i) \in [\![J]\!] \wedge (C', s', h', i) \in safe_{\mathsf{CSL}}(q, J)).\end{array}$$

We can then define:

$$\models C\ \mathbf{sat}_{\mathsf{CSL}}\ (p, q, J) \quad \stackrel{\text{def}}{=} \quad \forall (s, h, i) \in [\![p]\!].\, (C, s, h, i) \in safe_{\mathsf{CSL}}(q, J).$$

As an example of concurrent separation logic, consider the following program – adapted from one previously studied by O'Hearn [2004] – in which two threads communicate through a shared single-cell buffer at location `c`.

```
while (true) {                 ||  while (true) {
  x := alloc();                ||    when (full) atomic {
  when (!full) atomic {        ||      full := false;
    full := true;              ||      y := c;
    c := x;                    ||    }
  }                            ||    dispose(y);
}                              ||  }
```

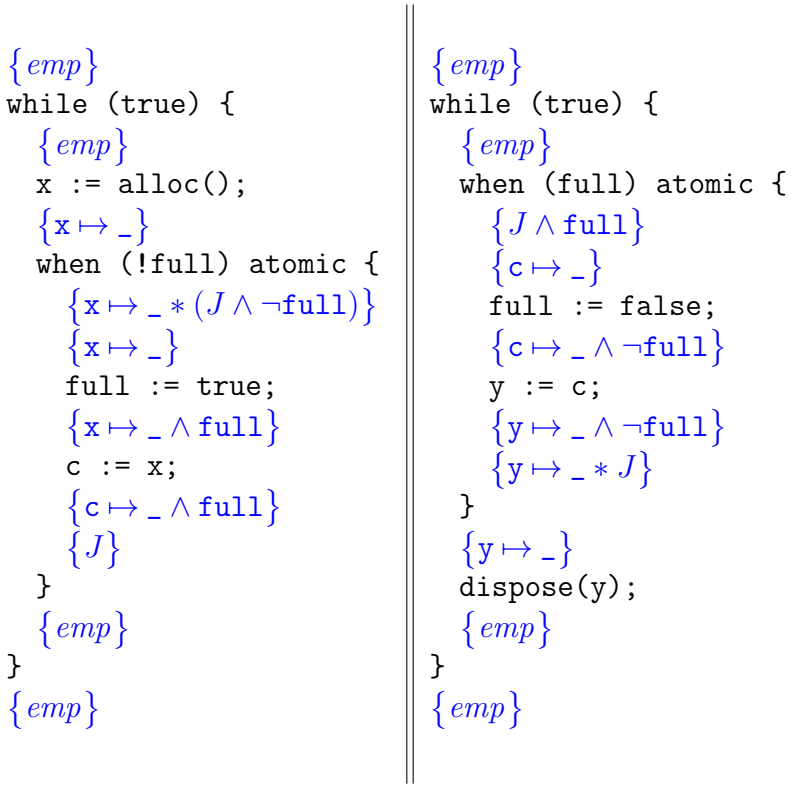


Figure 2.8: A proof outline in concurrent separation logic

We verify this program using concurrent separation logic in Fig. 2.8. The *resource invariant* is

$$J \stackrel{\text{def}}{=} (\mathtt{c} \mapsto \_ \wedge \mathtt{full}) \vee (\neg \mathtt{full} \wedge emp)$$

which means that the location `c` is shared exactly when the `full` flag is set. The left-hand 'producer' waits until the flag is unset to enter its critical region, during which it assigns to `c` the address of a newly created location. The right-hand 'consumer' thread waits until the flag is set to enter its critical region, during which it notes `c`'s new value.

2.8 Rely-guarantee

Rely-guarantee is an extension of Hoare Logic that enables compositional reasoning about concurrent programs. It was invented by Jones [1983]. The rely-guarantee method is similar to the Owicki-Gries method, but restores compositionality by putting more information into the judgements. These now take the following form:

$$\models C \ \mathbf{sat}_{\mathsf{RG}} \ (p, R, G, q).$$

In addition to the pre- and postcondition inherited from Hoare logic, a command is specified by two relations between states: a *rely* R that specifies all the atomic state transitions (called *actions*) the environment can cause, and a *guarantee* G that specifies all the actions of the command itself. (The environment is the set of concurrently-running threads.) Following Prensa Nieto [2003], G shall be reflexive. The command C satisfies the rely-guarantee specification (p, R, G, q) if whenever C begins execution in a state satisfying the precondition p, in an environment whose interference is limited to the actions in the rely R, then any state transitions

performed by C are within its guarantee G, and moreover, if the execution terminates, the final state satisfies the postcondition q.

Definition 2.16 (Meaning of judgements: rely-guarantee)**.** Let $safe_{\mathsf{RG}}(q, R, G)$ be the largest set containing only those triples (C, σ, i) that:

- do not fault:
$$(C, \sigma) \not\to \mathbf{abort},$$
- satisfy the postcondition q if they are terminal:
$$C = \mathtt{skip} \implies (\sigma, i) \in [\![q]\!],$$
- continue to satisfy these properties after any state transition in the rely R:
$$\forall \sigma'.\, (\sigma, \sigma') \in R \implies (C, \sigma', i) \in safe_{\mathsf{RG}}(q, R, G),$$
- and continue to satisfy these properties after any execution step (whose associated state transition must be in the guarantee G) by the current thread:
$$\forall C', \sigma'.\, (C, \sigma) \to (C', \sigma') \implies (\sigma, \sigma') \in G \wedge (C', \sigma', i) \in safe_{\mathsf{HL}}(q).$$

We can then define:

$$\models C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q) \quad \stackrel{\text{def}}{=} \quad \forall (\sigma, i) \in [\![p]\!].\, (C, \sigma, i) \in safe_{\mathsf{RG}}(q, R, G).$$

The rely-guarantee method conservatively assumes that between consecutive commands in a thread, any number of actions in R may occur. The truth of an assertion that holds after one command must be preserved by this interference, so that it may be safely assumed by the next command. Such an assertion is deemed *stable under* R.

Definition 2.17 (Stability)**.** If p is an assertion, and $R \subseteq \mathsf{PState} \times \mathsf{PState}$ is a relation between program states, then:

$$p\ \mathsf{stable\ under}\ R \quad \stackrel{\text{def}}{=} \quad \forall \sigma, \sigma', i.\, (\sigma, i) \in [\![p]\!] \wedge (\sigma, \sigma') \in R \implies (\sigma', i) \in [\![p]\!].$$

Figure 2.9 presents a selection of the rely-guarantee proof rules. The RG-BASIC rule requires that the basic command c meets the sequential specification $\{p\}\ c\ \{q\}$, and that any action it performs is within its guarantee. The RG-ATOMIC rule is similar. The pre- and postconditions of atomic blocks, basic commands and `skip` are all required to be stable. Since the other commands are built inductively from these, their rules can assume any inherited assertions to be stable (or else derived from stable assertions by the RG-CONSEQ rule).

The RG-PAR rule marks the epitome of rely-guarantee reasoning. When reasoning about commands composed in parallel, the rely of each command is extended to include the guarantee of the other. The composed command $C_1 \parallel C_2$ guarantees actions that are in the guarantees of either of its components, and establishes the postconditions of both of its components upon completion.

$$\textsc{RG-Conj}\ \frac{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_1, R, G, q_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_2, R, G, q_2)}{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_1 \wedge p_2, R, G, q_1 \wedge q_2)} \qquad \textsc{RG-Disj}\ \frac{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_1, R, G, q_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_2, R, G, q_2)}{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p_1 \vee p_2, R, G, q_1 \vee q_2)}$$

$$\textsc{RG-Conseq}\ \frac{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p', R', G', q') \qquad p \Rightarrow p' \qquad q' \Rightarrow q \qquad R \subseteq R' \qquad G' \subseteq G}{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)}$$

$$\textsc{RG-Par}\ \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{RG}}\ (p_1, R \cup G_2, G_1, q_1) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{RG}}\ (p_2, R \cup G_1, G_2, q_2)}{\vdash C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RG}}\ (p_1 \wedge p_2, R, G_1 \cup G_2, q_1 \wedge q_2)} \qquad \textsc{RG-Seq}\ \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, r) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{RG}}\ (r, R, G, q)}{\vdash C_1\ ;\ C_2\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)}$$

$$\textsc{RG-Basic}\ \frac{\vdash \{p\}\ c\ \{q\} \qquad \overleftarrow{[\![p]\!]} \cap \mathsf{transitions}(c) \subseteq G \qquad p\ \mathsf{stable\ under}\ R \qquad q\ \mathsf{stable\ under}\ R}{\vdash c\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)} \qquad \textsc{RG-Atomic}\ \frac{\vdash \{p\}\ C\ \{q\} \qquad (p \rightsquigarrow q) \subseteq G \qquad p\ \mathsf{stable\ under}\ R \qquad q\ \mathsf{stable\ under}\ R}{\vdash \mathtt{atomic}\, C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)}$$

$$\textsc{RG-Skip}\ \frac{p\ \mathsf{stable\ under}\ R}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, p)} \qquad \textsc{RG-Loop}\ \frac{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, p) \qquad p\ \mathsf{stable\ under}\ R}{\vdash \mathtt{loop}\, C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, p)}$$

Figure 2.9: Proof rules for rely-guarantee

$$\{\mathtt{x} = 0\}$$

$$\begin{array}{l|l} \{\mathtt{x} \geq 0\} & \{\mathtt{x} \geq 0\} \\ \mathtt{atomic\ \{\ x\ :=\ x+1\ \}} & \mathtt{atomic\ \{\ x\ :=\ x+1\ \}} \\ \{\mathtt{x} \geq 1\} & \{\mathtt{x} \geq 1\} \end{array}$$

$$\{\mathtt{x} \geq 1\}$$

Figure 2.10: Proof outline of parallel-increment in rely-guarantee

2.8.1 Example: parallel increment

We illustrate rely-guarantee reasoning by using it to verify a program comprising two threads, each of which atomically increments a variable x, which is initially zeroed. Like all basic commands, the increments happen instantaneously, but we wrap them in atomic blocks to make this clear. A first attempt is shown in Fig. 2.10. The final postcondition is weaker than the 'x = 2' assertion that one might expect. To explain this weakness, consider the right-hand thread. It is capable of incrementing x, so a suitable guarantee for it is

$$G_{\mathrm{rhs}} \stackrel{\mathrm{def}}{=} (\mathtt{x} = \overleftarrow{\mathtt{x}} + 1). \tag{2.6}$$

By RG-Par, this guarantee becomes the rely, R_{lhs}, of the left-hand thread. Now, in order to verify the basic command in the left-hand thread using the RG-Basic rule, its pre- and

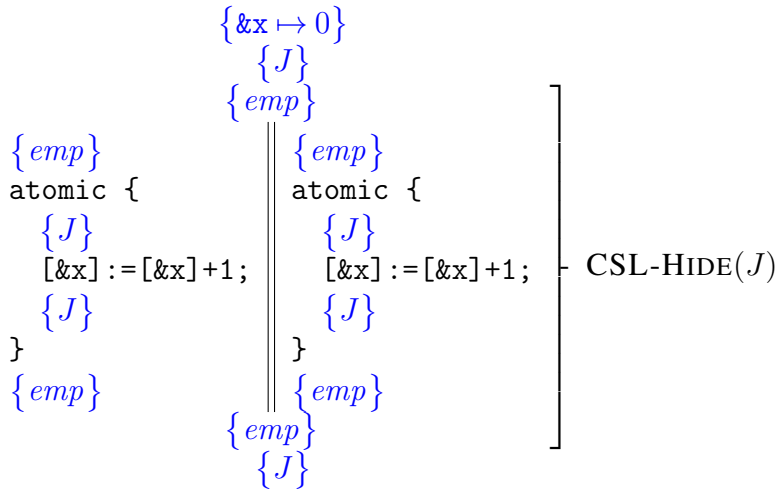


Figure 2.11: Proof outline of parallel-increment in concurrent separation logic

postcondition must be stable under R_{lhs}. Any assertion that specifies a particular value for x will certainly not be stable, as the environment can increase x at any time. Hence, once we enter the parallel composition, we must resort to weaker assertions that specify only a lower bound on the value of x.

It is pedagogical to repeat this exercise in concurrent separation logic. Concurrent separation logic does not permit variables to be shared between threads; it insists that all inter-thread communication happens via the heap instead. Accordingly, we shall replace the variable x with the heap location &x, for some constant x. We ought to split the increment into two commands – for reading the location and then writing it, in accordance with the programming language presented in Sect. 2.5 – but our meaning is clearer if we do not. The resource invariant J states that heap location &x contains a non-negative value; that is:

$$J \stackrel{\text{def}}{=} \exists n.\, \&\mathtt{x} \mapsto n \wedge n \geq 0.$$

The proof is shown in Fig. 2.11. At its outermost level, we use the CSL-HIDE rule to take the location &x out of the shared state so that it can feature in the overall pre- and postcondition. The postcondition obtained – that the value in location &x is non-negative – is even weaker than the postcondition we managed using rely-guarantee. This is an inherent limitation of invariant-based reasoning. Because the invariant J must hold both before and after the increment command, it is unable to distinguish whether the command has happened, not happened, or even happened many times.

2.8.2 Auxiliary code

There exists a fix that enables both concurrent separation logic and rely-guarantee to obtain the desired postcondition, $\mathtt{x} = 2$, for our parallel-increment example. The fix is to instrument our program with *auxiliary code*, as shown in Fig. 2.12. Auxiliary code is forbidden from affecting the behaviour of a program. It may read from variables used in the program, but it must not write to them. Hence, any judgement that holds for the instrumented program is immediately valid for the original program. Mindful that auxiliary code is sometimes called 'ghost code', we distinguish it from regular code by using a lighter colour. The purposes of the auxiliary program variables a and b are to record when the left and right threads, respectively, have performed their

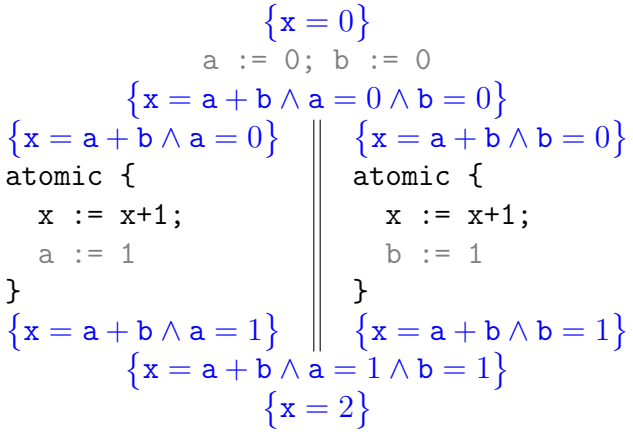


Figure 2.12: Proof outline of parallel-increment using rely-guarantee with auxiliary code

increments. The relies and guarantees for the left- and right-hand threads are now:

$$R_{\text{rhs}} \stackrel{\text{def}}{=} G_{\text{lhs}} \stackrel{\text{def}}{=} \begin{array}{l}((\overleftarrow{\mathtt{x}} = 0 \wedge \overleftarrow{\mathtt{a}} = 0 \wedge \overleftarrow{\mathtt{b}} = 0) \wedge (\mathtt{x} = 1 \wedge \mathtt{a} = 1 \wedge \mathtt{b} = 0)) \vee \\ ((\overleftarrow{\mathtt{x}} = 1 \wedge \overleftarrow{\mathtt{a}} = 0 \wedge \overleftarrow{\mathtt{b}} = 1) \wedge (\mathtt{x} = 2 \wedge \mathtt{a} = 1 \wedge \mathtt{b} = 1))\end{array}$$

$$R_{\text{lhs}} \stackrel{\text{def}}{=} G_{\text{rhs}} \stackrel{\text{def}}{=} \begin{array}{l}((\overleftarrow{\mathtt{x}} = 0 \wedge \overleftarrow{\mathtt{a}} = 0 \wedge \overleftarrow{\mathtt{b}} = 0) \wedge (\mathtt{x} = 1 \wedge \mathtt{a} = 0 \wedge \mathtt{b} = 1)) \vee \\ ((\overleftarrow{\mathtt{x}} = 1 \wedge \overleftarrow{\mathtt{a}} = 1 \wedge \overleftarrow{\mathtt{b}} = 0) \wedge (\mathtt{x} = 2 \wedge \mathtt{a} = 1 \wedge \mathtt{b} = 1)).\end{array}$$

These relations capture the fact that the threads do not actually need permission to increment x arbitrarily, as they had in (2.6). Consider the left-hand thread: it only increments x from 0 to 1 when the right-hand thread has not executed, and it only increments x from 1 to 2 when the right-hand thread has already executed. The auxiliary code enables this fine-grained reasoning about how the executions of the two threads can interleave.

Introduced by Clint [1973] and used extensively by Owicki and Gries [1976], the injection of auxiliary code remains central to several modern verification tools, such as VCC [Cohen et al. 2009] and VeriFast [Jacobs et al. 2011a]. However, we choose to avoid it in this dissertation. Jones [2010] explains that "[auxiliary variables] can be useful – but they can also be undesirable in that they can undermine the hard won property of compositionality". The problem is that when assembling a proof about a program composed of several already-verified threads, it may be necessary to instrument those threads with auxiliary code in order to finish the verification of the complete program; those threads must then be verified again. Jacobs and Piessens [2011] explain how this problem can be mitigated by having judgements parameterised by specifications of any auxiliary code that may be required. Their solution notwithstanding, a second problem with auxiliary code is that it can lead to complex proofs that cloud the intuitive reason for a program's correctness. Vafeiadis and Parkinson [2007] acknowledge that the verification of a non-blocking stack conducted by Parkinson et al. [2007] "requires a lot of auxiliary state to encode the possible interference", and that in their replacement logic, RGSep (which is the topic of our next section), "much of the auxiliary state can be removed, and hence the proof becomes clearer".

2.9 RGSep

RGSep is a combination of concurrent separation logic with rely-guarantee reasoning, due to Vafeiadis and Parkinson [2007]. It has several similarities to another program logic called

SAGL, which was developed concurrently but independently by Feng et al. [2007].

In concurrent separation logic, assertions in a thread describe only that part of the heap which is owned by that thread. At a program point outside of an atomic block, the shared heap appears not to exist. Within an atomic block, however, any location in either the shared heap or the thread-local heap becomes accessible.

In RGSep, threads may make assertions about both their local heap and the shared heap. To facilitate this, we update our notion of a program state so that the heap is divided into these two parts. A program state $\sigma \in \mathsf{PState}$ is now a triple (s, h, H) comprising a store s, a local heap h and a shared heap H, such that $h \uplus H$ is defined.

$$\mathsf{PState} \stackrel{\text{def}}{=} \{(s, h, H) \in \mathsf{Store} \times \mathsf{Heap} \times \mathsf{Heap} \mid \mathrm{dom}(h) \not\asymp \mathrm{dom}(H)\}$$

In fact, the actual heap may additionally contain locations in the local heaps of other threads, but since these are never visible, it is not necessary to include them in the notion of a state.

The assertion language of RGSep is as follows:[1]

$$P ::= p \mid \boxed{p} \mid P * P \mid P \wedge P \mid P \vee P \mid \exists x.\, P.$$

Ordinary separation logic assertions p refer to the thread's local state, while boxed assertions $\boxed{p}$ refer to the shared state. Because RGSep assertions include ordinary separation logic assertions, we shall temporarily introduce subscripts $[\![-]\!]_{\mathsf{RGSep}}$ and $[\![-]\!]_{\mathsf{SL}}$ in the following discussion of their meanings. We have:

$$\begin{array}{rcl} [\![p]\!]_{\mathsf{RGSep}} & = & \{(s, h, H, i) \mid (s, h, i) \in [\![p]\!]_{\mathsf{SL}} \wedge \mathrm{defined}(h \uplus H)\} \\ [\![\boxed{p}]\!]_{\mathsf{RGSep}} & = & \{(s, \emptyset, H, i) \mid (s, H, i) \in [\![p]\!]_{\mathsf{SL}}\}. \end{array}$$

The $*$-operator in RGSep is defined like so.

$$[\![P * P']\!]_{\mathsf{RGSep}} = \{(s, h \uplus h', H, i) \mid (s, h, H, i) \in [\![P]\!]_{\mathsf{RGSep}} \wedge (s, h', H, i) \in [\![P']\!]_{\mathsf{RGSep}}\}$$

The effect of this definition is that the $*$-operator separates local assertions, but acts like ordinary conjunction on boxed assertions, in order that all threads have the same view of the shared heap. In particular, we have the following property.

$$\boxed{p} * \boxed{q} = \boxed{p \wedge q} \tag{2.7}$$

A problem with allowing threads to make assertions about the shared heap is that the shared heap may change as a result of actions by other threads. This is where the rely-guarantee reasoning comes in. RGSep specifications contain a rely and a guarantee. The rely specifies all the changes to the shared state that other threads may make, while the guarantee specifies all the changes to the shared state that the current thread may make. Hence, it is safe for a thread to make an assertion $\boxed{p}$ about the shared state, providing $\boxed{p}$ is stable under the thread's rely.

In traditional rely-guarantee, the rely and the guarantee are relations over program states. In RGSep, these relations need not cover the local heap: the store and the shared heap suffice. That is, RGSep relies and guarantees have type

$$\mathcal{P}((\mathsf{Store} \times \mathsf{Heap}) \times (\mathsf{Store} \times \mathsf{Heap})).$$

[1] Following Dinsdale-Young et al. [2010], we depart from the original syntax of RGSep by swapping uppercase P and lowercase p here.

How such a relation is interpreted depends on whether it is a rely or a guarantee. If it is a guarantee, it simply contains all the state changes the current thread may cause. If it is a rely, say R, then the situation is more complex. Suppose the current program state is (s, h, H), and that the pair $((s, H), (s', H'))$ is in R. This means that the current program state may be changed to (s, h, H'). The store s and the local heap h remain unchanged because in RGSep interference can occur only via the shared heap. The transition can only occur if h and H' are disjoint, for only then is the resultant program state valid. Such conflicts between local and shared heaps do not feature in the original formulation of RGSep. We develop our refined notion of stability in the following two definitions.

Definition 2.18 (Rely restriction)**.** If $h \in \mathsf{Heap}$ is a (local) heap, and $R \in \mathcal{P}((\mathsf{Store} \times \mathsf{Heap}) \times (\mathsf{Store} \times \mathsf{Heap}))$ is a rely or a guarantee, then:

$$R \backslash h \stackrel{\text{def}}{=} \{((s, H), (s', H')) \in R \mid \text{defined}(h \uplus H) \wedge \text{defined}(h \uplus H')\}$$

Definition 2.19 (RGSep stability)**.** If P is an RGSep assertion, and $R \in \mathcal{P}((\mathsf{Store} \times \mathsf{Heap}) \times (\mathsf{Store} \times \mathsf{Heap}))$ is a rely or a guarantee, then:

$$\begin{aligned} P \text{ stable under } R \stackrel{\text{def}}{=} \forall s, h, H, H', i.\, (s, h, H, i) \in [\![P]\!]_{\mathsf{RGSep}} \wedge \\ (\exists s'.\, ((s, H), (s', H')) \in R \backslash h) \\ \implies (s, h, H', i) \in [\![P]\!]_{\mathsf{RGSep}}. \end{aligned}$$

RGSep can be understood as a generalisation of concurrent separation logic via the following encoding:

$$C \ \mathbf{sat}_{\mathsf{CSL}} \ (p, q, J) \quad = \quad C \ \mathbf{sat}_{\mathsf{RGSep}} \ (p * \boxed{J}, J \rightsquigarrow J, J \rightsquigarrow J, q * \boxed{J})$$

The encoding puts the shared state J into the pre- and postcondition. For both the rely and the guarantee, it uses the single action $J \rightsquigarrow J$. This means that any thread can modify the shared state providing the invariant J is preserved. Formally, such actions are defined below.

Definition 2.20 (RGSep actions)**.** The *action* $p \rightsquigarrow q$ denotes the following relation:

$$[\![p \rightsquigarrow q]\!] \stackrel{\text{def}}{=} \{((s, H \uplus H''), (s', H' \uplus H'')) \mid \exists i.\, (s, H, i) \in [\![p]\!]_{\mathsf{SL}} \wedge (s', H', i) \in [\![q]\!]_{\mathsf{SL}}\}.$$

The action replaces a part (H) of the shared heap satisfying p with a part (H') satisfying q. The rest of the shared heap (H'') is not affected.

Vafeiadis and Parkinson [2007] demonstate the power of the RGSep logic by verifying several fine-grained concurrent algorithms that cannot be handled by concurrent separation logic alone. Concurrent separation logic is based on *invariants* – predicates on a single state. In contrast, RGSep and rely-guarantee are based on *relations* – predicates on a pair of states. Where concurrent separation logic can only describe the ways in which the shared heap stays the same, RGSep can describe how it changes, and this is the key to its greater expressivity.

We now formally define the meaning of RGSep judgements.

Definition 2.21 (Meaning of judgements: RGSep)**.** Let $safe_{\mathsf{RGSep}}(Q, R, G)$ be the largest set containing only those quintuples (C, s, h, H, i) that:

- do not fault (even in the presence of a shared heap $H \in \mathsf{Heap}$ and extra heap locations h_o:
$$\forall h_o, h_1.\, h \uplus H \uplus h_o = h_1 \implies (C, (s, h_1)) \not\to \mathbf{abort},$$
- satisfy the postcondition Q if they are terminal:
$$C = \mathtt{skip} \implies (s, h, H, i) \in [\![Q]\!]_{\mathsf{RGSep}},$$
- continue to satisfy these properties after any shared-state transition in the rely R:
$$\forall H'.\, (\exists s'.\, ((s, H), (s', H')) \in R \backslash h) \implies (C, s, h, H', i) \in safe_{\mathsf{RGSep}}(Q, R, G),$$
- and continue to satisfy these properties after any execution step (whose associated shared-state transition must be in the guarantee G) by the current thread, noting that this execution step may occur in the presence of extra heap locations h_o that are unaffected by the step:
$$\begin{array}{l}\forall h_1, h_o, C', s', h'.\\ \quad h \uplus H \uplus h_o = h_1 \wedge (C, (s, h_1)) \to (C', (s', h'_1))\\ \quad \implies (\exists h', H'.\, h'_1 = h' \uplus H' \uplus h_o \wedge ((s, H), (s', H')) \in G\\ \qquad\qquad \wedge\, (C', s', h', H', i) \in safe_{\mathsf{RGSep}}(Q, R, G)).\end{array}$$

We can then define:

$$\models C\ \mathbf{sat}_{\mathsf{RGSep}}\ (P, R, G, Q) \stackrel{\text{def}}{=} \forall (s, h, H, i) \in [\![P]\!]_{\mathsf{RGSep}}.\ (C, s, h, H, i) \in safe_{\mathsf{RGSep}}(Q, R, G).$$

Figure 2.13 presents some proof rules for RGSep.

The RGSEP-PAR rule recalls the RG-PAR rule, but uses separating rather than ordinary conjunction in the pre- and postconditions. This allows the two threads to have separate local heaps, but the same shared heap.

The RGSEP-FRAME rule recalls separation logic's FRAME rule, but requires an additional side-condition that the frame F is stable under R and G. This is because F may describe the shared state, which is liable to be transformed in accordance with R or G during the execution of the command C.

The RGSEP-BASIC rule is simpler than rely-guarantee's RG-BASIC rule because RGSep forbids basic commands from accessing the shared state; they must be enclosed in an atomic block if they wish to do so. Since p and q describe only local state, it is not necessary to consider R or G.

In contrast, the RGSEP-ATOMIC rule is more complex than rely-guarantee's RG-ATOMIC rule. The side-conditions on stability under the rely and adherence to the guarantee are inherited from rely-guarantee. The precondition describes a local state satisfying p' and shared state satisfying $p * r$. The part of the shared state satisfying p is assimilated into the local state, then the command executes, leaving a local state satisfying $q' * q$. The part of this satisfying q is then

$$\textsc{RGSep-Par}\quad \frac{\vdash C_1\ \mathbf{sat}_{\mathsf{RGSep}}\ (P_1, R \cup G_2, G_1, Q_1) \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{RGSep}}\ (P_2, R \cup G_1, G_2, Q_2)}{\vdash C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RGSep}}\ (P_1 * P_2, R, G_1 \cup G_2, Q_1 * Q_2)}$$

$$\textsc{RGSep-Frame}\quad \frac{\vdash C\ \mathbf{sat}_{\mathsf{RGSep}}\ (P, R, G, Q) \qquad F \text{ stable under } (R \cup G)}{\vdash C\ \mathbf{sat}_{\mathsf{RGSep}}\ (P * F, R, G, Q * F)}$$

$$\textsc{RGSep-Basic}\quad \frac{\vdash \{p\}\ c\ \{q\}}{\vdash c\ \mathbf{sat}_{\mathsf{RGSep}}\ (p, R, G, q)}$$

$$\textsc{RGSep-Atomic}\quad \frac{\begin{array}{c}\vdash \{p * p'\}\ C\ \{q * q'\} \qquad P \Rightarrow \boxed{p * r} * p' \qquad \boxed{q * r} * q' \Rightarrow Q \\ p, q \text{ precise} \qquad P, Q \text{ stable under } R \qquad [\![p \rightsquigarrow q]\!] \subseteq G\end{array}}{\vdash \mathtt{atomic}\, C\ \mathbf{sat}_{\mathsf{RGSep}}\ (P, R, G, Q)}$$

Figure 2.13: Proof rules for RGSep

returned to the shared state, where it rejoins the remainder r that did not engage in the action. The side-condition concerning precision is a technical requirement that ensures the local/shared split is unambiguous – Vafeiadis [2007] provides the details.

2.10 Conclusion

This chapter has presented several logics for various types of sequential and concurrent programming languages. We have described only those that we draw upon in later chapters, so this is by no means an exhaustive survey. In Chapters 3 and 4 we shall focus on the extension and usage of the rely-guarantee and RGSep logics, while Chapter 5 builds on separation logic and variables-as-resource.

Chapter 3

Explicit stabilisation

This chapter proposes a new formalisation of stability for rely-guarantee reasoning, in which an assertion's stability is encoded into its syntactic form. It enables rely-guarantee, for the first time, to verify concurrent libraries independently of their clients' environments.

This chapter is based on a conference paper co-authored by Mike Dodds and Matthew Parkinson [Wickerson et al. 2010a].

In the previous chapter, we established the importance of stability of pre- and postconditions in rely-guarantee reasoning. Stability is traditionally enforced through side-conditions on proof rules. In the presentation of rely-guarantee by Prensa Nieto [2003], four of her six proof rules have stability checks, while Coleman and Jones [2007] make stability an implicit side-condition on every proof rule. This chapter proposes a new formalisation, in which stability is recorded within the syntactic form of the assertion itself, thus removing the need for these side-conditions. Just as 'explicit substitution' [Abadi et al. 1990] added substitution to the syntax of the λ-calculus, our work adds stabilisation to the syntax of rely-guarantee assertions. To this end, we propose two new constructions: $\lfloor p \rfloor_R$ to denote the weakest assertion that is both stronger than p and stable under R, and $\lceil p \rceil_R$ to denote the strongest assertion that is both weaker than p and stable under R.

The following five benefits provide the motivation for this work.

Lazy stabilisation Explicit stabilisation brings benefits for automatic tools – such as SmallfootRG [Calcagno et al. 2007] – that employ rely-guarantee proof rules, by allowing stability to be evaluated lazily. Rather than stabilising at the point of applying a proof rule, we can use our new notation to record that the assertion must be stable, and carry on. Later, we may choose either to evaluate the stabilised term or, if that part of the proof is abandoned, to discard it. Often, a single assertion is required to be stabilised several times – perhaps it is used as the postcondition of one command and as the precondition of the next – and in such cases, we can exploit the properties of our stabilisation operators to deduce that these stabilisations can be collapsed together. We have not considered the implementation of stabilisation, but this issue is explored by Amjad and Bornat [2009].

Simplification of complex rely-guarantee rules Coleman [2008] proposes a proof rule for reasoning about conditional statements whose test conditions are evaluated in the presence of environmental interference. Section 3.2 describes how his rule, which relies on subtle arguments about stability, can be both simplified and generalised with the help of explicit stabilisation.

Verification of concurrent libraries Rely-guarantee is a compositional method. That is, an entire program's proof depends only upon the proofs of its constituent commands. It is not, however, modular. That is, a command's proof cannot necessarily be reused when the command features in a different program. This is because proofs are environment-specific. As a result, rely-guarantee cannot, in general, verify libraries that are invoked in several different environments. Section 3.3 explains how explicit stablisation can rectify this situation by delaying the evaluation of stability until the client's environment is known. We require the library to record stability requirements using $\lfloor - \rfloor_R$ and $\lceil - \rceil_R$ and to leave the specification parametric in R. Each client then instantiates R appropriately and performs the stabilisation.

Encoding of several variant proof systems Vafeiadis [2007] proposes four variations of the rely-guarantee proof system, each of which imposes stability checks at different places, and each of which has subtly different expressiveness. Section 3.4 explains how to encode these four variations as instances of a single, general proof system that employs explicit stabilisation.

Information hiding In the next chapter, we explain how explicit stabilisation allows sequential modules to hide 'internal interference' from their clients.

3.1 Explicit stabilisation for rely-guarantee

This section describes explicit stabilisation and how it is applied to the rely-guarantee proof rules. The properties presented in this section have been formalised using the Isabelle theorem prover. The proof script is available from the following webpage.

`http://www.cl.cam.ac.uk/~jpw48/expstab.thy.html`

We propose two new syntactic constructs, named *stable floor* and *stable ceiling*. The stable floor, written $\lfloor p \rfloor_R$, is the weakest assertion that is stronger than p and stable under R; that is:

$$\lfloor p \rfloor_R \Rightarrow p \tag{3.1}$$

$$\lfloor p \rfloor_R \text{ stable under } R \tag{3.2}$$

$$\text{for all } q, \text{ if } q \Rightarrow p \text{ and } q \text{ stable under } R \text{ then } q \Rightarrow \lfloor p \rfloor_R. \tag{3.3}$$

The stable ceiling, written $\lceil p \rceil_R$, is the strongest assertion that is weaker than p and stable under R; that is:

$$p \Rightarrow \lceil p \rceil_R \tag{3.4}$$

$$\lceil p \rceil_R \text{ stable under } R \tag{3.5}$$

$$\text{for all } q, \text{ if } p \Rightarrow q \text{ and } q \text{ stable under } R \text{ then } \lceil p \rceil_R \Rightarrow q. \tag{3.6}$$

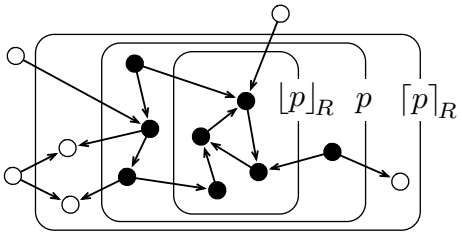
Constructions defined by the properties above have previously been used by Vafeiadis [2007] while defining the semantics of his RGSep judgements. Our contribution is to put these constructions into the syntax of assertions.

Properties (3.1) to (3.6) are realised by the following constructions. In fact, these constructions are unique, so we henceforth take them as the definitions of the stable floor and ceiling.

Definition 3.1 (Semantics of explicit stabilisation)**.**

$$\begin{aligned}\llbracket \lfloor p \rfloor_R \rrbracket &\stackrel{\text{def}}{=} \{\sigma \mid \forall \sigma'.\, (\sigma, \sigma') \in R^* \Rightarrow \sigma' \in \llbracket p \rrbracket\} \\ \llbracket \lceil p \rceil_R \rrbracket &\stackrel{\text{def}}{=} \{\sigma \mid \exists \sigma'.\, (\sigma', \sigma) \in R^* \wedge \sigma' \in \llbracket p \rrbracket\}\end{aligned}$$

The picture below presents the intuition behind our new operators. The nodes represent states; those that are filled satisfy some assertion p. The edges depict transitions of an arbitrary rely R. The states in $\lfloor p \rfloor_R$ are those from which any reachable state satisfies p. The states in $\lceil p \rceil_R$ are those reachable from a state in p.



Our operators can also be defined using the predicate transformer semantics due to Dijkstra [1976]: $\lfloor p \rfloor_R$ is the weakest precondition of R^* given postcondition p, while $\lceil p \rceil_R$ is the strongest postcondition of R^* given precondition p.

Example 3.2. Let $R_{\mathtt{x}+} \stackrel{\text{def}}{=} (\overleftarrow{\mathtt{x}} \leq \mathtt{x})$. That is, $R_{\mathtt{x}+}$ is a rely representing an environment that can always increase the value of $\mathtt{x}$. The assertions $\mathtt{x} = 0$ and $\mathtt{x} \neq 0$ are stabilised under $R_{\mathtt{x}+}$ as follows:

$$\begin{aligned}\lfloor \mathtt{x} = 0 \rfloor_{R_{\mathtt{x}+}} &\Leftrightarrow \mathit{false} \\ \lceil \mathtt{x} = 0 \rceil_{R_{\mathtt{x}+}} &\Leftrightarrow \mathtt{x} \geq 0 \\ \lfloor \mathtt{x} \neq 0 \rfloor_{R_{\mathtt{x}+}} &\Leftrightarrow \mathtt{x} > 0 \\ \lceil \mathtt{x} \neq 0 \rceil_{R_{\mathtt{x}+}} &\Leftrightarrow \mathit{true}.\end{aligned}$$

3.1.1 Properties

Both $\lfloor - \rfloor_R$ and $\lceil - \rceil_R$ are monotonic with respect to $\Rightarrow$.

$$p \Rightarrow q \quad \text{implies} \quad \lfloor p \rfloor_R \Rightarrow \lfloor q \rfloor_R \tag{3.7}$$
$$p \Rightarrow q \quad \text{implies} \quad \lceil p \rceil_R \Rightarrow \lceil q \rceil_R . \tag{3.8}$$

They are related via the equivalence

$$\lfloor \neg p \rfloor_R \quad \Leftrightarrow \quad \neg \lceil p \rceil_{R^{-1}} . \tag{3.9}$$

Each has no effect if (and only if) its operand is already stable.

$$p \text{ stable under } R \quad \text{iff} \quad \lfloor p \rfloor_R \Leftrightarrow p \quad \text{iff} \quad \lceil p \rceil_R \Leftrightarrow p. \tag{3.10}$$

In particular, stabilisation has no effect when the rely is empty. Both $true$ and $false$ are stable, and conjunction and disjunction both preserve stability.

$$true \text{ stable under } R \tag{3.11}$$
$$false \text{ stable under } R \tag{3.12}$$
$$p \text{ stable under } R \text{ and } q \text{ stable under } R \text{ implies } p \wedge q \text{ stable under } R \tag{3.13}$$
$$p \text{ stable under } R \text{ and } q \text{ stable under } R \text{ implies } p \vee q \text{ stable under } R. \tag{3.14}$$

Regarding distributivity over conjunction and disjunction: the stable floor has the same behaviour as the $\forall$-quantifier, while the stable ceiling recalls the $\exists$-quantifier.

$$\lfloor p \wedge q \rfloor_R \;\Leftrightarrow\; \lfloor p \rfloor_R \wedge \lfloor q \rfloor_R \tag{3.15}$$
$$\lfloor p \vee q \rfloor_R \;\Leftarrow\; \lfloor p \rfloor_R \vee \lfloor q \rfloor_R \tag{3.16}$$
$$\lceil p \wedge q \rceil_R \;\Rightarrow\; \lceil p \rceil_R \wedge \lceil q \rceil_R \tag{3.17}$$
$$\lceil p \vee q \rceil_R \;\Leftrightarrow\; \lceil p \rceil_R \vee \lceil q \rceil_R \,. \tag{3.18}$$

Several properties mirror those of the floor and ceiling functions in arithmetic, from which our syntax is borrowed. If $R \subseteq R'$, we have:

$$\lfloor \lfloor p \rfloor_R \rfloor_{R'} \;\Leftrightarrow\; \lfloor \lfloor p \rfloor_{R'} \rfloor_R \;\Leftrightarrow\; \lceil \lfloor p \rfloor_{R'} \rceil_R \;\Leftrightarrow\; \lfloor p \rfloor_{R'} \tag{3.19}$$
$$\lceil \lceil p \rceil_R \rceil_{R'} \;\Leftrightarrow\; \lceil \lceil p \rceil_{R'} \rceil_R \;\Leftrightarrow\; \lfloor \lceil p \rceil_{R'} \rfloor_R \;\Leftrightarrow\; \lceil p \rceil_{R'} \tag{3.20}$$

As the rely gets larger in order to accommodate additional environmental interference, stability becomes harder to show. That is:

$$R \subseteq R' \text{ implies } \lfloor p \rfloor_R \Leftarrow \lfloor p \rfloor_{R'} \tag{3.21}$$
$$R \subseteq R' \text{ implies } \lceil p \rceil_R \Rightarrow \lceil p \rceil_{R'} \,. \tag{3.22}$$

Our stabilisation operators can be understood as Galois adjoints. Let $\mathsf{StableAssertion}(R)$ be the set of assertions that are stable under R. Then $\lfloor - \rfloor_R$ is the upper adjoint of the embedding from $\mathsf{StableAssertion}(R)$ into $\mathsf{Assertion}$, and $\lceil - \rceil_R$ is its lower adjoint. That is:

$$\lceil p \rceil_R \Rightarrow q \text{ iff } p \Rightarrow q \tag{3.23}$$
$$q \Rightarrow \lfloor p \rfloor_R \text{ iff } q \Rightarrow p \tag{3.24}$$

for all $p \in \mathsf{Assertion}$ and $q \in \mathsf{StableAssertion}(R)$.

3.1.2 Application to rely-guarantee proof rules

We explain how to adapt the rely-guarantee proof rules from Fig. 2.9 to use explicit stabilisation rather than side-conditions.

The rules in Fig. 3.1 are the replacements for the RG-BASIC, RG-ATOMIC, RG-SKIP and RG-LOOP rules. The others remain unchanged. The new set of rules is at least as powerful as the original set. Indeed, the original rules can easily be obtained by restoring the stability checks and then erasing the redundant stabilisations.

The RG-BASIC-S rule first derives precondition p and postcondition q by considering c sequentially; that is, without concern for stability. A concurrent specification is then obtained by strengthening p and weakening q until they are both stable.

$$\text{RG-BASIC-S}\quad \frac{\vdash \{p\}\ c\ \{q\} \qquad \{(\sigma,\sigma') \mid \sigma \in [\![p]\!] \wedge \sigma' \in [\![c]\!](\sigma)\} \subseteq G}{\vdash c\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_R, R, G, \lceil q \rceil_R)}$$

$$\text{RG-ATOMIC-S}\quad \frac{\vdash \{p\}\ C\ \{q\} \qquad (p \rightsquigarrow q) \subseteq G}{\vdash \mathtt{atomic}\, C\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_R, R, G, \lceil q \rceil_R)} \qquad \text{RG-SKIP-S}\quad \frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, \lceil p \rceil_R)}$$

$$\text{RG-LOOP-S}\quad \frac{\vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, p)}{\vdash \mathtt{loop}\, C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, \lceil p \rceil_R)}$$

Figure 3.1: Proof rules for rely-guarantee (with explicit stabilisation)

The RG-SKIP-S rule has several equivalent alternatives. The following rule is intended for backward reasoning.

$$\text{RG-SKIP-S2}\quad \frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_R, R, G, p)}$$

Also equivalent are the following axioms.

$$\text{RG-SKIP-S3}\quad \frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p \rceil_R, R, G, \lceil p \rceil_R)} \qquad \text{RG-SKIP-S4}\quad \frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_R, R, G, \lfloor p \rfloor_R)}$$

Lemma 3.3. *The* RG-SKIP*,* RG-SKIP-S*,* RG-SKIP-S2*,* RG-SKIP-S3 *and* RG-SKIP-S4 *rules are all equivalently powerful.*

Proof.

- RG-SKIP-S3 is obtained from RG-SKIP by instantiating p to $\lceil p \rceil_R$, and then noting that the stability check in RG-SKIP becomes redundant, by (3.10).
- RG-SKIP-S is obtained from RG-SKIP-S3 by strengthening the precondition from $\lceil p \rceil_R$ to p.
- RG-SKIP-S4 is obtained from RG-SKIP-S by instantiating p to $\lfloor p \rfloor_R$, and then noting that the stable ceiling can be removed, by (3.19).
- RG-SKIP-S2 is obtained from RG-SKIP-S4 by weakening the postcondition from $\lfloor p \rfloor_R$ to p.
- RG-SKIP is obtained from RG-SKIP-S2 by noting that the stable floor can be removed under the assumption that p is stable under R, by (3.19). □

3.2 Simplifying complex rely-guarantee proof rules

We highlight the elegance of explicit stabilisation by showing how it can simplify and generalise complex rely-guarantee proof rules that rely subtly upon stability.

Coleman [2008] considers reasoning about one-armed conditional statements that are evaluated in the presence of environmental interference. His work is motivated by programs such as those of the form

$$\texttt{if } (\texttt{x} = 0 \wedge \texttt{y} = 0) \texttt{ then } \mathit{body} \tag{3.25}$$

running in an environment that can set x to any positive value, and may not modify y. When *body* begins execution, we know that x and y must have evaluated to 0, but we do not know that x is still 0. Coleman proposes the following proof rule, which imposes that only the stable conjunct $\texttt{y} = 0$ can be safely used as a precondition for *body*. (We have simplified the rule by assuming, unlike Coleman, that expressions are evaluated atomically.)

$$\frac{\begin{array}{c}\mathsf{StableExpr}(e_s, R) \\ \vdash C \ \mathbf{sat}_{\mathsf{RG}}\ (p \wedge e_s, R, G, q) \\ \vdash \texttt{skip}\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, \neg(e_s \wedge e_u) \Rightarrow q) \\ \{\neg e_u, p, q\}\ \mathsf{stable\ under}\ R\end{array}}{\vdash \texttt{if } e_u \wedge e_s \texttt{ then } C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)}$$

Test conditions are of the form $e_u \wedge e_s$: the conjunction of an unstable assertion and a stable assertion. The first premise requires the stable conjunct e_s to contain no variables that R can change. Of the two conjuncts, only e_s can be assumed still to hold when C begins execution (second premise). The third premise requires that if the test fails, the postcondition is met without evaluating C. (Having embraced the relational calculus for postconditions, as described in Sect. 2.2.1, Coleman can express this premise more succinctly as $\overleftarrow{p} \wedge R^* \wedge \neg(e_s \wedge e_u) \Rightarrow q$.) The fourth premise requires R to preserve the falsity of e_u. This ensures that the obligation to fulfil q cannot be bypassed in the case when the test initially evaluates to false but later becomes logically true.

Now consider the following alternative rule, which uses explicit stabilisation. It has fewer and simpler premises, and it handles two-armed conditionals easily. Moreover, the test condition need not be split into stable and unstable conjuncts.

$$\frac{\begin{array}{c}\{p, q\}\ \mathsf{stable\ under}\ R \\ \vdash C_1\ \mathbf{sat}_{\mathsf{RG}}\ (p \wedge \lceil e \rceil_R, R, G, q) \\ \vdash C_2\ \mathbf{sat}_{\mathsf{RG}}\ (p \wedge \lceil \neg e \rceil_R, R, G, q)\end{array}}{\vdash \texttt{if } e \texttt{ then } C_1 \texttt{ else } C_2\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q)}$$

The execution of C_1 begins in a state satisfying $\lceil e \rceil_R$; that is, a state that is reachable (by a sequence of environment actions) from one in which e evaluated to true. Similarly, $\lceil \neg e \rceil_R$ describes a state reached from one where e did not hold. Stability checks on p and q remain for compatibility with the rest of Coleman's system.

Revisiting our example from (3.25), we find that by taking the stable ceiling of the test condition, rather than simply removing the unstable parts, we can obtain a stronger precondition for the *body* command, namely $\texttt{x} \geq 0 \wedge \texttt{y} = 0$.

3.3 Explicit stabilisation and library verification

We explain why the traditional rely-guarantee method is unsuitable for reasoning about concurrent libraries. Essentially, the problem is that rely-guarantee specifications involve a particular rely. This means that a specification can only be used in

a context whose rely is the same as (or smaller than) that in the specification. To improve the reusability of specifications, we introduce a 'parametric rely-guarantee' proof system, which is underpinned by explicit stabilisation.

In this section we are concerned with the problem of verifying concurrent library code using the rely-guarantee method. The central challenge of library verification is to produce not merely a specification for the library, but one that can be used to verify any client.

The traditional rely-guarantee method is inherently unsuitable for reasoning about libraries. The problem is that each rely-guarantee specification mentions a particular rely, and hence can only be used in a context whose rely is the same as (or smaller than) that in the specification. Flanagan et al. [2005] have previously applied the rely-guarantee method to the verification of programs comprising several procedures, but their procedure specifications still involve a particular rely, which limits their reuse.

That the rely-guarantee method is unsuitable for reasoning about concurrent libraries is plainly witnessed by its failure to provide a reusable specification even for one of the most trivial functions that one might find in a library: increment.

Consider a library function `f()` that atomically increments a shared variable `x`. Consider also two of its clients, `g()` and `h()`, which invoke `f()` in an empty environment and an environment that may increase `x`, respectively. Call this latter environment $R_{\mathtt{x+}}$. The guarantee $G_{\mathtt{x+}}$ additionally dictates that no variable other than `x` changes.

Definition 3.4.
$$\begin{aligned}
\mathtt{f()} &\stackrel{\text{def}}{=} \mathtt{x := x + 1} \\
\mathtt{g()} &\stackrel{\text{def}}{=} \mathtt{assume}(\mathtt{x} = 3)\ ;\ \mathtt{f()}\ ;\ \mathtt{assert}(\mathtt{x} = 4) \\
\mathtt{h()} &\stackrel{\text{def}}{=} \mathtt{assume}(\mathtt{x} = 5)\ ;\ (\mathtt{f()} \parallel \mathtt{f()})\ ;\ \mathtt{assert}(\mathtt{x} \geq 6) \\
R_{\mathtt{x+}} &\stackrel{\text{def}}{=} \overleftarrow{\mathtt{x}} \leq \mathtt{x} \\
G_{\mathtt{x+}} &\stackrel{\text{def}}{=} \overleftarrow{\mathtt{x}} \leq \mathtt{x} \wedge \forall y \neq \mathtt{x}.\overleftarrow{y} = y
\end{aligned}$$

Our ability to verify both `g()` and `h()` hinges, respectively, upon obtaining the following two specifications for `f()`:

$$\mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\mathtt{x} = X, \emptyset, G_{\mathtt{x+}}, \mathtt{x} = X + 1) \qquad \mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\mathtt{x} \geq X, R_{\mathtt{x+}}, G_{\mathtt{x+}}, \mathtt{x} \geq X + 1)$$

Both of these judgements hold. However, there exists no single 'most general' specification from which both can be derived. Note that the first specification has a stronger postcondition than the second, but the second has a larger rely, and neither is derivable from the other via the RG-CONSEQ rule. If a 'most general' specification were to exist, it would have a large rely, and also a strong postcondition. Yet there is a tension that puts these requirements at odds with each other. The larger the rely, the tougher the stability requirement. The tougher the stability requirement, the more the postcondition must be weakened in order for it to become stable.

Note that the postcondition of `h()` is weaker than the '$\mathtt{x} = 7$' one might expect. As explained in Sect. 2.8.1, this limitation is unavoidable unless auxiliary code is employed. We choose to avoid auxiliary code because, as explained in Sect. 2.8.2, it can undermine compositionality and lead to complex proofs.

We shall now develop a proof system that enables a 'most general' specification for `f()` to be found. The specification will be parameterised by a rely, and this parameter will be instantiated on a per-client basis. The specification will state that the postcondition of `f()` needs weakening from '$\mathtt{x} = X + 1$' just enough to become stable under whichever rely is chosen by the client.

When this rely is $R_{\mathtt{x+}}$ – that is, when verifying `g()` – the postcondition becomes '$\mathtt{x} \geq X + 1$'. And when the rely is $\emptyset$ – that is, when verifying `h()` – no weakening is required.

This "weakening [...] just enough to become stable" is an instance of our stable ceiling operator. We can use it to express the 'most general' specification described above like so:

$$\text{for any rely } R\text{:}\quad \mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil \mathtt{x} = X \rceil_R, R, G_{\mathtt{x+}}, \lceil \mathtt{x} = X + 1 \rceil_R). \tag{3.26}$$

In truth, the situation is slightly more subtle, as there is a restriction on which values R can take. Before going into these details, we shall formalise a proof system, called *parametric rely-guarantee*, that works with specifications, such as (3.26), that are quantified over a set of relies.

A parametric rely-guarantee specification takes the form

$$C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})$$

Compared to an ordinary rely-guarantee specification, the rely has become a set of relies (ranged over by $\mathbf{R}$), and the pre- and postcondition are drawn from the set ParamAssertion of *parametric assertions* (ranged over by $\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$), which is defined by the following grammar.

$$\begin{array}{lll}
\mathbf{p} ::= \lambda R.\, p & & \text{(basic parametric assertion)} \\
\quad \mid \mathbf{p} \vee \mathbf{p} \mid \mathbf{p} \wedge \mathbf{p} & & \text{(disjunction; conjunction)} \\
\quad \mid \exists x.\, \mathbf{p} & & \text{(existential quantification over a logical variable)} \\
\quad \mid \lfloor \mathbf{p} \rfloor & & \text{(stable floor)} \\
\quad \mid \lceil \mathbf{p} \rceil & & \text{(stable ceiling)}
\end{array}$$

Note that the stable floor and ceiling no longer need to be parameterised by a rely now that *all* assertions are parameterised by a rely. Where an ordinary assertion denotes a set of states, a parametric assertion denotes a function from relies to sets of states. We use λ-calculus notation to describe basic parametric assertions; note that $\lambda\_.\, p$ denotes the constant p. The meanings of the other constructions in ParamAssertion are given by lifting the meanings of ordinary assertions to functions in the natural way. To this end, we define the following translation function:

$$\begin{array}{rcl}
\langle\!| - |\!\rangle_{_} & : & \mathsf{ParamAssertion} \to \mathcal{P}(\mathsf{PState} \times \mathsf{PState}) \to \mathsf{Assertion} \\
\langle\!| \lambda R.\, p |\!\rangle_R & \stackrel{\text{def}}{=} & p \\
\langle\!| \mathbf{p}_1 \vee \mathbf{p}_2 |\!\rangle_R & \stackrel{\text{def}}{=} & \langle\!| \mathbf{p}_1 |\!\rangle_R \vee \langle\!| \mathbf{p}_2 |\!\rangle_R \\
\langle\!| \mathbf{p}_1 \wedge \mathbf{p}_2 |\!\rangle_R & \stackrel{\text{def}}{=} & \langle\!| \mathbf{p}_1 |\!\rangle_R \wedge \langle\!| \mathbf{p}_2 |\!\rangle_R \\
\langle\!| \exists x.\, \mathbf{p} |\!\rangle_R & \stackrel{\text{def}}{=} & \exists x.\, \langle\!| \mathbf{p} |\!\rangle_R \\
\langle\!| \lfloor \mathbf{p} \rfloor |\!\rangle_R & \stackrel{\text{def}}{=} & \lfloor \langle\!| \mathbf{p} |\!\rangle_R \rfloor_R \\
\langle\!| \lceil \mathbf{p} \rceil |\!\rangle_R & \stackrel{\text{def}}{=} & \lceil \langle\!| \mathbf{p} |\!\rangle_R \rceil_R
\end{array}$$

The translation function is used in the following definition of the semantics of parametric rely-guarantee judgements.

Definition 3.5 (Meaning of judgements: parametric rely-guarantee)**.** A parametric rely-guarantee specification represents a family of rely-guarantee specifications, one for each rely in $\mathbf{R}$.

$$\models C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q}) \stackrel{\text{def}}{=} \forall R \in \mathbf{R}.\, \models C\ \mathbf{sat}_{\mathsf{RG}}\ (\langle\!| \mathbf{p} |\!\rangle_R, R, G, \langle\!| \mathbf{q} |\!\rangle_R)$$

PRG-CONJ
$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_1, \mathbf{R}, G, \mathbf{q}_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_2, \mathbf{R}, G, \mathbf{q}_2)}{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_1 \wedge \mathbf{p}_2, \mathbf{R}, G, \mathbf{q}_1 \wedge \mathbf{q}_2)}$$

PRG-DISJ
$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_1, \mathbf{R}, G, \mathbf{q}_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_2, \mathbf{R}, G, \mathbf{q}_2)}{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_1 \vee \mathbf{p}_2, \mathbf{R}, G, \mathbf{q}_1 \vee \mathbf{q}_2)}$$

PRG-CONSEQ
$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}', \mathbf{R}', G', \mathbf{q}') \qquad \models \mathbf{p} \Rightarrow_{\mathbf{R}} \mathbf{p}' \qquad \models \mathbf{q}' \Rightarrow_{\mathbf{R}} \mathbf{q} \qquad \mathbf{R} \subseteq \mathbf{R}' \qquad G' \subseteq G}{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})}$$

PRG-BASIC
$$\frac{\vdash \{p\}\ c\ \{q\} \qquad \overleftarrow{[\![p]\!]} \cap \mathsf{transitions}(c) \subseteq G}{\vdash c\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\ \lfloor p \rfloor_R, \mathbb{U}, G, \lambda R.\ \lceil q \rceil_R)}$$

PRG-ATOMIC
$$\frac{\vdash \{p\}\ C\ \{q\} \qquad (p \rightsquigarrow q) \subseteq G}{\vdash \mathtt{atomic}(C)\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\ \lfloor p \rfloor_R, \mathbb{U}, G, \lambda R.\ \lceil q \rceil_R)}$$

PRG-EXISTS
$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})}{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\exists x.\ \mathbf{p}, \mathbf{R}, G, \exists x.\ \mathbf{q})}$$

PRG-SKIP
$$\frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \lceil \mathbf{p} \rceil)}$$

PRG-LOOP
$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{p})}{\vdash \mathtt{loop}\, C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \lceil \mathbf{p} \rceil)}$$

PRG-CHOICE
$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q}) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})}{\vdash C_1\ \mathtt{or}\ C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})}$$

PRG-SEQ
$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{r}) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{r}, \mathbf{R}, G, \mathbf{q})}{\vdash C_1\ ;\ C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q})}$$

PRG-PAR
$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_1, \mathbf{R} \cup G_2, G_1, \mathbf{q}_1) \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_2, \mathbf{R} \cup G_1, G_2, \mathbf{q}_2)}{\vdash C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}_{1\ G_2} \parallel_{G_1} \mathbf{p}_2, \mathbf{R}, G_1 \cup G_2, \mathbf{q}_{1\ G_2} \parallel_{G_1} \mathbf{q}_2)}$$

Figure 3.2: Proof rules for parametric rely-guarantee

The proof rules for parametric rely-guarantee are presented in Fig. 3.2. The rules use the following abbreviations:

$$\begin{aligned} \models \mathbf{p}_1 \Rightarrow_{\mathbf{R}} \mathbf{p}_2 &\stackrel{\text{def}}{=} \forall R \in \mathbf{R}.\ \models \langle\!| \mathbf{p}_1 |\!\rangle_R \Rightarrow \langle\!| \mathbf{p}_2 |\!\rangle_R \\ \mathbf{R} \cup R &\stackrel{\text{def}}{=} \{R' \cup R \mid R' \in \mathbf{R}\} \\ \mathbb{U} &\stackrel{\text{def}}{=} \text{the universal set of all relies, } \mathcal{P}(\mathsf{PState} \times \mathsf{PState}). \end{aligned}$$

Several of the rules closely resemble the original non-parametric rules. One notable departure is the PRG-PAR rule, which has grown considerably more complex. The rule uses a '$\mathbf{p}_{1\ R'} \parallel_{R''} \mathbf{p}_2$' operator, which is translated like so:

$$\langle\!| \mathbf{p}_{1\ R'} \parallel_{R''} \mathbf{p}_2 |\!\rangle_R \stackrel{\text{def}}{=} \langle\!| \mathbf{p}_1 |\!\rangle_{R \cup R'} \wedge \langle\!| \mathbf{p}_2 |\!\rangle_{R \cup R''}.$$

The idea is that at the fork and join of parallel commands, the rely changes. If the rely is R initially, then within the component commands the rely becomes either $R \cup G_2$ or $R \cup G_1$. After the join, it reverts to R. Our rule simply reflects this progression.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{p\}\ \mathtt{x := x+1}\ \{p[\mathtt{x}-1/\mathtt{x}]\}}\ \text{Floyd's assignment axiom}}{\vdash \{\lceil \mathtt{x}=X\rceil_R\}\ \mathtt{x := x+1}\ \{\lceil \mathtt{x}=X\rceil_R[\mathtt{x}-1/\mathtt{x}]\}}\ \text{Instantiate } p \text{ to } \lceil \mathtt{x}=X\rceil_R}{\vdash \mathtt{x := x+1}\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\lceil \mathtt{x}=X\rceil_R, \mathbb{U}, G_{\mathtt{x}+}, \lambda R.\lceil\lceil \mathtt{x}=X\rceil_R[\mathtt{x}-1/\mathtt{x}]\rceil_R)}\ \text{PRG-BASIC}}{\vdash \mathtt{x := x+1}\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\lceil \mathtt{x}=X\rceil_R, \mathsf{comm}_{\mathtt{x}+}, G_{\mathtt{x}+}, \lambda R.\lceil \mathtt{x}=X+1\rceil_R)}\ \text{PRG-CONSEQ}$$

Figure 3.3: Derivation of parametric specification for `f()`

The PRG-ATOMIC, PRG-BASIC and PRG-SKIP rules deduce specifications that feature the universal set of relies. This allows them to be deployed in *any* environment. The PRG-CONSEQ rule can then be used to shrink this set. Doing so restricts a specification's reusability, but it enhances the applicability of the $\Rightarrow_{\mathbf{R}}$ relation, through which it can be simplified.

This tradeoff between reusability and simplicity is illustrated by Fig. 3.3, which derives a parametric specification for `f()`. The derivation begins with an instance of the forwards-reasoning assignment axiom due to Floyd [1967]. In applying the PRG-BASIC rule, we utilise the identity $\lfloor\lceil \mathtt{x}=X\rceil_R\rfloor_R \Leftrightarrow \lceil \mathtt{x}=X\rceil_R$, which is an instance of (3.20). The specification on the third line is the most general, as it allows the rely to be instantiated freely. Yet we do not stop there. We restrict the rely to the set $\mathsf{comm}_{\mathtt{x}+}$ of those that 'commute' with the increment operation; that is, for which

$$\lceil p\rceil_R[\mathtt{x}-1/\mathtt{x}] \quad\Leftrightarrow\quad \lceil p[\mathtt{x}-1/\mathtt{x}]\rceil_R \tag{3.27}$$

holds for all p. Using (3.27), we can simplify the postcondition. We thus arrive at the specification envisaged in (3.26), albeit with the extra condition that the rely is in the $\mathsf{comm}_{\mathtt{x}+}$ set. The specification in (3.26) is actually invalid without this condition. (For instance, if the rely represents, say, an environment that can only increment `x` from 2 to 3:

$$R_{23} \stackrel{\text{def}}{=} \overleftarrow{\mathtt{x}} = 2 \wedge \mathtt{x} = 3$$

and if the logical variable X is 2, then the specification in (3.26) becomes

$$\mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\mathtt{x}=2 \vee \mathtt{x}=3, R_{23}, G_{\mathtt{x}+}, \mathtt{x}=3)$$

which is false.) The diagram below shows informally how the parametric specification can then be instantiated to two ordinary specifications, for use in proving the two clients `g()` and `h()`.

$$\vdash \mathtt{f()}\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\lceil \mathtt{x}=X\rceil_R, \mathsf{comm}_{\mathtt{x}+}, G_{\mathtt{x}+}, \lambda R.\lceil \mathtt{x}=X+1\rceil_R)$$

$$\text{Set } R \text{ to } \emptyset \swarrow \qquad \searrow \text{Set } R \text{ to } R_{\mathtt{x}+}$$

$$\vdash \mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\mathtt{x}=X, \emptyset, G_{\mathtt{x}+}, \mathtt{x}=X+1) \qquad \vdash \mathtt{f()}\ \mathbf{sat}_{\mathsf{RG}}\ (\mathtt{x}\geq X, R_{\mathtt{x}+}, G_{\mathtt{x}+}, \mathtt{x}\geq X+1)$$

Really, this 'instantiation' is an application of the PRG-CONSEQ rule to restrict $\mathbf{R}$ to the singletons $\{\emptyset\}$ and $\{R_{\mathtt{x}+}\}$ respectively.

We observe that the 'most general' specifications that our parametric scheme can deduce are, though sometimes desirable, inhibited by their complexity. The specification on the third line of Fig. 3.3 contains two stabilisation operations in its postcondition – and this is for just a single basic command. For the program that increments `x` twice consecutively, we can obtain the following 'most general' specification:

$$\vdash \begin{pmatrix}\mathtt{x := x+1;}\\ \mathtt{x := x+1}\end{pmatrix}\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\lceil \mathtt{x}=X\rceil_R, \mathbb{U}, G_{\mathtt{x}+}, \lambda R.\lceil\lceil\lceil \mathtt{x}=X\rceil_R[\mathtt{x}-1/\mathtt{x}]\rceil_R[\mathtt{x}-1/\mathtt{x}]\rceil_R).$$

Under the assumption that the rely commutes with the increment operation, we can simplify the postcondition like before, to obtain a pleasingly simple and highly reusable specification for a library procedure that comprises more than a single atomic operation:

$$\vdash \begin{pmatrix} \mathtt{x := x + 1;} \\ \mathtt{x := x + 1} \end{pmatrix} \mathbf{sat}_{\mathsf{PRG}}\ (\lambda R.\ \lceil \mathtt{x} = X \rceil_R, \mathsf{comm}_{\mathtt{x+}}, G_{\mathtt{x+}}, \lambda R.\ \lceil \mathtt{x} = X + 2 \rceil_R).$$

For a sequence of n basic commands, the 'most general' specification may require up to $n + 1$ stabilisation operations in the postcondition, to model the environmental interference before, between and after each command. The complexity of the specification is thus comparable to the implementation it describes. Accordingly, it is crucial that our scheme allows specifications to be specialised to restricted sets of relies, and thence, simplified.

Theorem 3.6. *The proof rules of parametric stability are sound. That is:*

$$\vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q}) \implies \models C\ \mathbf{sat}_{\mathsf{PRG}}\ (\mathbf{p}, \mathbf{R}, G, \mathbf{q}).$$

Theorem 3.7. *The ordinary rely-guarantee proof rules of Fig. 2.9 (in which assertions do not contain explicit stabilisation), can be encoded completely and soundly in parametric rely-guarantee. That is:*

$$\begin{aligned} \vdash C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q) &\implies \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda\_.\, p, \mathcal{P}(R), G, \lambda\_.\, q) \\ \models C\ \mathbf{sat}_{\mathsf{RG}}\ (p, R, G, q) &\impliedby \models C\ \mathbf{sat}_{\mathsf{PRG}}\ (\lambda\_.\, p, \mathcal{P}(R), G, \lambda\_.\, q). \end{aligned}$$

In the encoding given above, the use of powersets lets the PRG-CONSEQ rule emulate the RG-CONSEQ rule.

3.4 Early, mid and late stability

We describe three variants of the rely-guarantee proof system, each due to Vafeiadis [2007]. These variants differ according to where in the proof rules the stability checks are located. We show that each variant can be encoded in our parametric rely-guarantee proof system.

The RG-ATOMIC and RG-BASIC rules both check that the pre- and postconditions are stable. Yet, in a sequence of atomic blocks and basic commands such as

$$\mathtt{atomic}\, C_1\ ;\ c_2\ ;\ \mathtt{atomic}\, C_3\ ;\ \ldots$$

the postcondition of one command is the same as the precondition of the next. Hence, almost half of these stability checks are redundant. Vafeiadis [2007] proposes three different schemes that eliminate these redundant checks through judicious placement of the stability side-conditions. In *early stability*, the checks are immediately after each atomic block or basic command, and at the forks of parallel composition. In *late stability*, the checks are immediately before each atomic block or basic command, and at the joins of parallel composition. In *mid stability*, the checks are at the sequencing operator, and at the forks and joins of parallel composition.

In the following informal illustration of the four schemes applied to a simple program, we indicate stability checks with a dot, and assume that C_1 through C_6 are atomic blocks or basic

commands.

ORIGINAL SCHEME

$$\bullet C_1 \bullet \;;\; \frac{(\bullet C_2 \bullet \;;\; \bullet C_3 \bullet)}{(\bullet C_4 \bullet \;;\; \bullet C_5 \bullet)} \;;\; \bullet C_6 \bullet$$

EARLY STABILITY

$$C_1 \bullet \;;\; \frac{\bullet(C_2 \bullet \;;\; C_3 \bullet)}{\bullet(C_4 \bullet \;;\; C_5 \bullet)} \;;\; C_6 \bullet$$

LATE STABILITY

$$\bullet C_1 \;;\; \frac{(\bullet C_2 \;;\; \bullet C_3)\bullet}{(\bullet C_4 \;;\; \bullet C_5)\bullet} \;;\; \bullet C_6$$

MID STABILITY

$$C_1 \bullet\!\!;\; \frac{\bullet(C_2 \bullet\!\!;\; C_3)\bullet}{\bullet(C_4 \bullet\!\!;\; C_5)\bullet} \bullet\!\!;\; C_6$$

An immediate problem is that the early stability scheme does not check stability of the final postcondition, nor does the late stability scheme check stability of the first precondition. The mid stability scheme omits both stability checks. The resolution is to customise the meaning of judgements for each scheme, as follows.

Definition 3.8 (Meaning of judgements: early, mid and late stability)**.**

$$\begin{aligned}
\models C\ \mathbf{sat}_{\mathsf{RG\text{-}E}}\ (p, R, G, q) &\stackrel{\text{def}}{=} \forall R' \subseteq R. \models C\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_{R'}, R', G, \lfloor q \rfloor_{R'}) \\
\models C\ \mathbf{sat}_{\mathsf{RG\text{-}L}}\ (p, R, G, q) &\stackrel{\text{def}}{=} \forall R' \subseteq R. \models C\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p \rceil_{R'}, R', G, \lceil q \rceil_{R'}) \\
\models C\ \mathbf{sat}_{\mathsf{RG\text{-}M}}\ (p, R, G, q) &\stackrel{\text{def}}{=} \forall R' \subseteq R. \models C\ \mathbf{sat}_{\mathsf{RG}}\ (\lfloor p \rfloor_{R'}, R', G, \lceil q \rceil_{R'})
\end{aligned}$$

In the semantics of early stability, we take the stable floor of the precondition; this serves to ensure that the first precondition in the main thread is stable. The stable floor in the postcondition serves no purpose. For late stability, the situation is mirrored: we require the final postcondition to be stable so we take its stable ceiling. As before, the stable ceiling in the precondition is redundant. Mid stability uses the stable floor in the precondition and the stable ceiling in the postcondition. (The missing fourth possible combination, with the stable ceiling in the precondition and the stable floor in the postcondition, actually gives rise to the same proof rules as in the original scheme.) All three definitions involve quantification over all stronger relies. This is an unfortunate inelegance that arises because the RG-CONSEQ rule allows the rely to be strengthened during the proof.

The full set of proof rules for early, mid and late stability are given in Fig. 3.4. Compared to the original presentation of these rules by Vafeiadis [2007], we have included some new rules for conjoining and disjoining specifications. Of these, the RG-CONJ-M, RG-CONJ-L, RG-DISJ-E and RG-DISJ-M rules require some stability checks as a result of the irregular distributivity of stabilisation over conjunction and disjunction (properties (3.15)–(3.18)).

Theorem 3.9 (Soundness of early, mid and late stability)**.**

$$\begin{aligned}
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}E}}\ (p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}E}}\ (p, R, G, q) \\
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}L}}\ (p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}L}}\ (p, R, G, q) \\
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}M}}\ (p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}M}}\ (p, R, G, q)
\end{aligned}$$

Proof. By rule induction. For instance, to show the soundness of the RG-PAR-L rule, we assume

$$\begin{aligned}
&\forall R' \subseteq R \cup G_2. \models C_1\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \rceil_{R'}, R', G_1, \lceil q_1 \rceil_{R'}) \\
&\forall R' \subseteq R \cup G_1. \models C_2\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_2 \rceil_{R'}, R', G_2, \lceil q_2 \rceil_{R'})
\end{aligned}$$

RG-CONJ-(E/M/L)

$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1, R, G, q_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_2, R, G, q_2) \quad \text{M/L only: } q_1, q_2 \text{ stable under } R}{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1 \wedge p_2, R, G, q_1 \wedge q_2)}$$

RG-DISJ-(E/M/L)

$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1, R, G, q_1) \quad \vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_2, R, G, q_2) \quad \text{E/M only: } p_1, p_2 \text{ stable under } R}{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1 \vee p_2, R, G, q_1 \vee q_2)}$$

RG-CONSEQ-(E/M/L)

$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p', R', G', q') \quad p \Rightarrow p' \quad q' \Rightarrow q \quad R \subseteq R' \quad G' \subseteq G}{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, q)}$$

RG-SEQ-(E/M/L)

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, r) \quad \vdash C_2\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (r, R, G, q) \quad \text{M only: } r \text{ stable under } R}{\vdash C_1\ ;\ C_2\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, q)}$$

RG-PAR-(E/M/L)

$$\frac{\vdash C_1\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1, R \cup G_2, G_1, q_1) \quad \text{E/M only: } p_1 \text{ stable under } R \cup G_2 \quad \text{M/L only: } q_1 \text{ stable under } R \cup G_2 \qquad \vdash C_2\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_2, R \cup G_1, G_2, q_2) \quad \text{E/M only: } p_2 \text{ stable under } R \cup G_1 \quad \text{M/L only: } q_2 \text{ stable under } R \cup G_1}{\vdash C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p_1 \wedge p_2, R, G_1 \cup G_2, q_1 \wedge q_2)}$$

RG-BASIC-(E/M/L)

$$\frac{\vdash \{p\}\ c\ \{q\} \quad \overleftarrow{[\![p]\!]} \cap \mathsf{transitions}(c) \subseteq G \quad \text{L only: } p \text{ stable under } R \quad \text{E only: } q \text{ stable under } R}{\vdash c\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, q)}$$

RG-ATOMIC-(E/M/L)

$$\frac{\vdash \{p\}\ C\ \{q\} \quad (p \rightsquigarrow q) \subseteq G \quad \text{L only: } p \text{ stable under } R \quad \text{E only: } q \text{ stable under } R}{\vdash \mathtt{atomic}\, C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, q)}$$

RG-SKIP-(E/M/L)

$$\frac{}{\vdash \mathtt{skip}\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, p)}$$

RG-LOOP-(E/M/L)

$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, p) \quad \text{M only: } p \text{ stable under } R}{\vdash \mathtt{loop}\, C\ \mathbf{sat}_{\mathsf{RG\text{-}(E/M/L)}}\ (p, R, G, p)}$$

Figure 3.4: Proof rules for early, mid and late stability

or equivalently

$$\forall R' \subseteq R.\ \models C_1\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \rceil_{R' \cup G_2}, R' \cup G_2, G_1, \lceil q_1 \rceil_{R' \cup G_2}) \tag{3.28}$$

$$\forall R' \subseteq R.\ \models C_2\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_2 \rceil_{R' \cup G_1}, R' \cup G_1, G_2, \lceil q_2 \rceil_{R' \cup G_1}) \tag{3.29}$$

and must prove, for arbitrary $R' \subseteq R$, that

$$\models C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \wedge p_2 \rceil_{R'}, R', G_1 \cup G_2, \lceil q_1 \wedge q_2 \rceil_{R'}). \tag{3.30}$$

By (3.28), (3.29) and the soundness of the RG-PAR rule, we have

$$\models C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \rceil_{R' \cup G_2} \wedge \lceil p_2 \rceil_{R' \cup G_1}, R', G_1 \cup G_2, \lceil q_1 \rceil_{R' \cup G_2} \wedge \lceil q_2 \rceil_{R' \cup G_1}).$$

Strengthening the precondition in accordance with (3.22), we obtain

$$\models C_1 \parallel C_2\ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \rceil_{R'} \wedge \lceil p_2 \rceil_{R'}, R', G_1 \cup G_2, \lceil q_1 \rceil_{R' \cup G_2} \wedge \lceil q_2 \rceil_{R' \cup G_1}).$$

Then, strengthening the precondition in accordance with (3.17), we obtain

$$\models C_1 \parallel C_2 \ \mathbf{sat}_{\mathsf{RG}}\ (\lceil p_1 \wedge p_2 \rceil_{R'}, R', G_1 \cup G_2, \lceil q_1 \rceil_{R' \cup G_2} \wedge \lceil q_2 \rceil_{R' \cup G_1}). \tag{3.31}$$

To handle the postcondition, we note that since q_1 and q_2 are stable under $R \cup G_2$ and $R \cup G_1$ respectively, we have $\lceil q_1 \rceil_{R' \cup G_2} \Leftrightarrow q_1$ and $\lceil q_2 \rceil_{R' \cup G_1} \Leftrightarrow q_2$ by (3.10), hence

$$\lceil q_1 \rceil_{R' \cup G_2} \wedge \lceil q_2 \rceil_{R' \cup G_1} \Leftrightarrow q_1 \wedge q_2.$$

Also, q_1 and q_2 are stable under R', hence so is $q_1 \wedge q_2$ by (3.13), so $q_1 \wedge q_2 \Leftrightarrow \lceil q_1 \wedge q_2 \rceil_{R'}$. We can hence replace the postcondition in (3.31) with the one in (3.30). This completes this case of the induction. □

We are able to encode all four variant proof systems into the parametric proof system presented in the previous section. The encoding of the original scheme has already been presented in Thm. 3.7. The encodings of early, mid and late stability specifications are as follows:

$$\begin{aligned}
\mathcal{E}(p, R, G, q) &\stackrel{\text{def}}{=} (\lambda R.\, \lfloor p \rfloor_R, \mathcal{P}(R), G, \lambda R.\, \lfloor q \rfloor_R) \\
\mathcal{L}(p, R, G, q) &\stackrel{\text{def}}{=} (\lambda R.\, \lceil p \rceil_R, \mathcal{P}(R), G, \lambda R.\, \lceil q \rceil_R) \\
\mathcal{M}(p, R, G, q) &\stackrel{\text{def}}{=} (\lambda R.\, \lfloor p \rfloor_R, \mathcal{P}(R), G, \lambda R.\, \lceil q \rceil_R)
\end{aligned}$$

The following theorems state that each encoding is sound and complete.

Theorem 3.10 (Completeness of encodings)**.**

$$\begin{aligned}
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}E}}\ (p, R, G, q) &\implies \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{E}(p, R, G, q) \\
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}L}}\ (p, R, G, q) &\implies \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{L}(p, R, G, q) \\
\vdash C\ \mathbf{sat}_{\mathsf{RG\text{-}M}}\ (p, R, G, q) &\implies \vdash C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{M}(p, R, G, q)
\end{aligned}$$

Proof. By rule induction on the $\mathbf{sat}_{\mathsf{RG\text{-}E}}$, $\mathbf{sat}_{\mathsf{RG\text{-}L}}$ and $\mathbf{sat}_{\mathsf{RG\text{-}M}}$ proof rules. □

Theorem 3.11 (Soundness of encodings)**.**

$$\begin{aligned}
\models C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{E}(p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}E}}\ (p, R, G, q) \\
\models C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{L}(p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}L}}\ (p, R, G, q) \\
\models C\ \mathbf{sat}_{\mathsf{PRG}}\ \mathcal{M}(p, R, G, q) &\implies \models C\ \mathbf{sat}_{\mathsf{RG\text{-}M}}\ (p, R, G, q)
\end{aligned}$$

Proof. By unfolding Defns. 3.5 and 3.8. □

Chapter 4

Explicit stabilisation and sequential modules

Originally conceived for concurrency verification, we explain why RGSep is also useful for reasoning about sequential modules. We show how explicit stabilisation can be applied to RGSep, and explain how this enables a proof rule for simple modules that hides some details of the module's implementation while verifying its clients. As a case study, we verify the memory manager from Version 7 Unix.

This chapter is based on a conference paper co-authored by Mike Dodds and Matthew Parkinson [Wickerson et al. 2010a].

4.1 Reasoning about modules

In his PhD dissertation, Vafeiadis [2007] remarks that

> reasoning about concurrent programs is very similar to reasoning about modular sequential programs. A sequential program with modules is essentially a coarse-grained concurrent program with one lock per module. Verification of these two classes of programs raises almost the same issues. The practical difference is that the difficult issues appear much earlier in concurrent programs than they do in modular sequential programs.

This chapter puts this part of Vafeiadis' thesis to the test, by applying his RGSep logic to the verification of a sequential module that interacts with its clients in subtle ways. Our case study is the memory manager from Version 7 Unix [Bell Labs 1979], but our results apply equally to a broad range of sequential modules.

One way to understand the connection between concurrent programs and sequential modules to which Vafeiadis alludes is that both require *abstraction*. In a concurrent setting, the actions of other threads must be abstracted in order to avoid the combinatorial explosion that would result from trying to consider all the possible interleavings of instructions from several threads. In the setting of a sequential module, the actions of clients must be abstracted simply because the client may not exist yet.

The central challenge in the verification of any module is to expose to clients a specification that reveals minimal details about the module's implementation (so that the implementation could later be changed without having to re-verify clients) while still revealing enough information to enable clients to be successfully verified. In this dissertation, we shall follow O'Hearn

et al. [2004] and deem a module simply to be a collection of functions that have privileged access to a particular region of the state. We shall call this region the 'module state'.

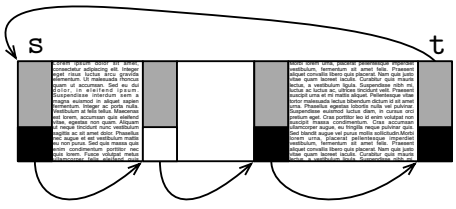
We are interested in heap-manipulating programs, so separation logic is the sensible basis for our verification. O'Hearn et al. [2004] and Parkinson and Bierman [2005] propose two different ways of using separation logic to verify modules. The former uses the *hypothetical frame rule*, while the latter uses *abstract predicates*. Roughly, the former *removes* the module state from the client's view, while the latter merely *obscures* it. Both sets of authors use simple memory managers as case studies. Although our memory manager can be considered a rather straightforward combination of those studied by O'Hearn et al. and by Parkinson and Bierman, we shall find that verifying it requires more than a straightforward combination of their approaches.

In the next subsections, we describe the implementation of the Version 7 Unix memory manager, then how it can be specified, and finally how we can verify that the implementation meets the desired specification.

4.1.1 How the Version 7 Unix memory manager works

Responsibility for allocating and deallocating memory ultimately lies with the operating system, but it is inefficient for clients to make memory requests to the system directly. The memory manager sits between the client and the operating system, and maintains a cache (called the 'arena') of available memory chunks, from which it tries to fulfil clients' requests. When a client frees a chunk of memory, it is not returned directly to the system, but instead held in this arena. The memory manager only resorts to a system call for more memory once it has exhausted its arena.

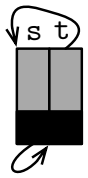
In the case of the Version 7 Unix memory manager, the arena is constructed from a chain of monotonically-increasing pointers (coloured grey in the picture below), which starts at `s` and ends at `t`. The gaps between these pointers are called chunks, and each is either allocated to a client or available. In the picture, the first and third chunks are currently allocated. Chunks are word-aligned, which means that the lower bits of each pointer are redundant. The least significant of these is thus employed as a 'busy' bit, and is set when the chunk following it is allocated. In the picture, the busy bit is black when set and white otherwise. The last pointer points back to the start of the arena, and because it is not followed by an allocatable chunk, its busy bit is permanently set.



A client requests a chunk of `nb` bytes by calling `malloc(nb)`. The routine traverses pointers until it finds a free chunk that is sufficiently large, returning the null pointer in the case of failure. It coalesces consecutive free chunks during the search. If the chunk it finds is larger than necessary, then any remainder is split off to form a new chunk. The client can later invoke `free(ap)` to hand the chunk beginning at `ap` back to the memory manager. All this routine needs to do is to unset the busy bit of the pointer preceding `ap`.

$\{uninit\}$
`u := malloc(2 * WORD);`
$\{mod\_inv * token\,\mathtt{u}\,2 * \boxed{\mathtt{u}}\}$
`v := malloc(3 * WORD);`
$\{mod\_inv * token\,\mathtt{u}\,2 * \boxed{\mathtt{u}} * token\,\mathtt{v}\,3 * \boxed{\mathtt{v}}\}$
`[u+1]:=5;`
$\{mod\_inv * token\,\mathtt{u}\,2 * \boxed{\mathtt{u}}{\scriptstyle 5} * token\,\mathtt{v}\,3 * \boxed{\mathtt{v}}\}$
`free(u);`
$\{mod\_inv * token\,\mathtt{v}\,3 * \boxed{\mathtt{v}}\}$
`free(v);`
$\{mod\_inv\}$

Figure 4.1: Proof outline of a simple client using the specifications in (4.1)



Regarding initialisation: the memory manager begins with just two consecutive cells, at locations `s` and `s`$+1$. When `malloc` is invoked for the first time, these two cells are formed into an initial arena. This is done by making them point to each other, setting their busy bits, and assigning `t` to be `s` $+ 1$, as shown in the picture to the right.

4.1.2 Specifying the memory manager

Parkinson and Bierman [2005, §3.3.2] argue that the following specifications are natural and useful descriptions of the `malloc` and `free` procedures.

$$\begin{array}{l}\left\{\begin{array}{l} mod\_inv \\ \vee\ uninit \end{array}\right\}\ \mathtt{malloc(nb)}\ \left\{\begin{array}{l} mod\_inv * token\,\mathtt{ret}\ \lceil \mathtt{nb/WORD} \rceil \\ * \boxed{\mathtt{ret}}^{\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil} \end{array}\right\} \\ \left\{mod\_inv * token\,\mathtt{ap}\,n * \boxed{\mathtt{ap}}^{\mathtt{ap}+n}\right\}\ \mathtt{free(ap)}\ \{mod\_inv\}\end{array} \tag{4.1}$$

We have changed some names, and added the *uninit* (for 'uninitialised') predicate to account for the fact that the first invocation of `malloc` initialises the arena. Figure 4.1 shows these specifications in action, being used to verify a simple client of the memory manager.

The various components of the specifications are as follows.

- The assertion $\boxed{\mathtt{ret}}^{\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil}$ describes the block of $\lceil \mathtt{nb/WORD} \rceil$ cells that are made available to the caller of `malloc`. Note that the ceiling notation here denotes arithmetic rounding rather than explicit stabilisation. The rounding is necessary because the memory is requested in bytes, but allocated in words.

- Also given to that caller is a token. This token must be surrendered when making the corresponding call to `free`, and certifies that the cells being returned were genuinely allocated by the memory manager. Without such a certificate, the client could pass to `free` an arbitrary sequence of memory cells, in violation of the acceptable behaviour laid down in the Unix manual [Kernighan and McIlroy 1979]. Note that tokens cannot be created, modified or duplicated by clients; this is because *token* is an abstract predicate, which is to say that its definition is not visible to clients.

- The module invariant mod_inv describes the module state.

Remark 4.1. We choose to leave the $uninit$ and mod_inv predicates separate, because this will later allow us to treat mod_inv as an abstract predicate. However, we could remove the disjunction in `malloc`'s precondition by absorbing $uninit$ into the definition of mod_inv; that is, by redefining the module invariant to encompass those states where the arena is uninitialised.

If we do so, the mod_inv predicate becomes ever-present throughout the pre- and postconditions. It is also protected from direct access by clients, and as such, it is sensible to remove it altogether from the client-facing specifications, using O'Hearn et al.'s hypothetical frame rule. Doing so would yield the following specifications.

$$\{emp\}\ \texttt{malloc(nb)}\ \left\{token\ \texttt{ret}\ \lceil \texttt{nb/WORD} \rceil * \left|\begin{array}{|l}\texttt{ret}\\\hline\end{array}\right|^{\texttt{ret}+\lceil \texttt{nb/WORD} \rceil}\right\}$$
$$\left\{token\ \texttt{ap}\ n * \left|\begin{array}{|l}\texttt{ap}\\\hline\end{array}\right|^{\texttt{ap}+n}\right\}\ \texttt{free(ap)}\ \{emp\}$$

For the time being, however, we shall not seek to remove mod_inv. We shall find that once we have found interpretations for mod_inv and $token$, it will no longer be clear that the hypothetical frame rule remains applicable. Hence, we shall leave the removal of mod_inv as an option to be explored in future work.

In order to verify that a particular implementation of `malloc` and `free` fits the specification in (4.1), we must find suitable definitions for $token$ and mod_inv. O'Hearn et al. do not need tokens because they consider a simplified memory manager that only doles out chunks with a fixed size of two cells. Parkinson and Bierman's memory manager has variable-sized chunks, so they do require tokens; however, their memory manager keeps no internal state, so the module invariant is not needed. Our memory manager works with variable-sized chunks, and maintains an internal state, hence we require non-trivial definitions for both $token$ and mod_inv. In the next subsection we shall explain how to find such definitions.

4.1.3 Verifying the memory manager

Describing the arena

Here is our first attempt at a formal definition of a valid arena. First, we define the following shorthand for setting the busy bit of a pointer.

$$x_{\blacksquare} \stackrel{\text{def}}{=} x + \tfrac{1}{\texttt{WORD}}$$

Next, we give predicates $chunk_{\mathsf{u}}$ and $chunk_{\mathsf{a}}$ that describe, respectively, unallocated and allocated chunks. Note that for each binary relation r, we define $x\,\dot{r}\,y$ to abbreviate $(x\,r\,y) \wedge emp$.

$$chunk_{\mathsf{u}}\,x\,y \stackrel{\text{def}}{=} \left|\begin{array}{|l}x\\\hline y\end{array}\right|^{y} \tag{4.2}$$

$$chunk_{\mathsf{a}}\,x\,y \stackrel{\text{def}}{=} x \mathrel{\dot{<}} y * \left|\begin{array}{|l}x\\\hline y_{\blacksquare}\end{array}\right| \tag{4.3}$$

The entire contents of an arena is summarised by a list C. Each element of such a list is a triple $\langle x, \tau, y\rangle$; it describes a chunk whose pointer is located at x and points to y, and has busy status $\tau \in \{\mathsf{u}, \mathsf{a}\}$. From a client's perspective, such a chunk comprises $y - x - 1$ usable cells, the first of which is at location $x+1$. In order for this list to be well-formed, the chunks must be in order and must not overlap; that is, for each $\langle x, \tau, y\rangle$ in C we have $x < y$, and for each consecutive pair $\langle x, \tau, y\rangle, \langle x', \tau', y'\rangle$ we have $y \leq x'$. We shall concatenate two lists C and C' by writing $C \circ C'$, and we shall write $C \circ\!\!- C'$ for the list C'' which is such that $C = C' \circ C''$. Either of

these operations may yield 'undefined', thus falsifying that fragment of the assertion in which it appears. That is, following Gordon et al. [1979], we treat $\Phi(e_1, \ldots, e_n)$, where Φ is an n-ary predicate on expressions, as being short for

$$\Phi(e_1, \ldots, e_n) \wedge \mathrm{defined}(e_1) \wedge \ldots \wedge \mathrm{defined}(e_n).$$

A series of chunks is described by the prediate *chunks*, defined below. The entire arena, described by the *arena* predicate, comprises a series of chunks from `s` to `t`, plus a pointer looping back from `t` to `s`. An uninitialised arena comprises just the cells at `s` and `s` + 1, both zeroed.

$$\begin{aligned} \mathit{chunks}\ x\ y\ C &\stackrel{\text{def}}{=} (x \doteq y * C \doteq [\,]) \vee \\ &\quad \exists z, \tau.\ \mathit{chunk}_\tau\ x\ z * \mathit{chunks}\ z\ y\ (C \circ\!\!- [\langle x, \tau, z\rangle]) \end{aligned} \tag{4.4}$$

$$\mathit{arena}\ A \stackrel{\text{def}}{=} \exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{t}\ C * A \doteq C^{\mathtt{a}} * \left|\begin{smallmatrix}\mathtt{t}\\ \mathtt{s}_\blacksquare\end{smallmatrix}\right| \tag{4.5}$$

$$\mathit{uninit} \stackrel{\text{def}}{=} \left|\begin{smallmatrix}s & \\ 0 & 0\end{smallmatrix}\right| \tag{4.6}$$

In the *arena* predicate given above, we expose the parameter A, which is an unordered list of chunks in the arena that are known to be allocated. Each element of this list is a mapping $(x \mapsto n)$, where x is the first cell in the chunk and n is the number of cells in the chunk. The list A is thus a partial function, and can be obtained by projecting all the allocated chunks from C. Note that in A, the first component of each element is the first cell in the chunk, but that in C, in the first component of each element is the pointer that *precedes* the first cell in the chunk. This projection is defined like so:

$$C^{\mathtt{a}} \stackrel{\text{def}}{=} \{(x + 1 \mapsto y - x - 1) \mid \langle x, \mathtt{a}, y\rangle \in C\}.$$

Defining the token predicate and the module invariant

We must find implementations for the *mod_inv* and *token* predicates that appear in the specifications in (4.1). Here are our first attempts.

$$\mathit{mod\_inv} \stackrel{\text{def}}{=} \exists A.\ \mathit{arena}\ A \tag{4.7}$$

$$\mathit{token}\ x\ n \stackrel{\text{def}}{=} \exists A.\ \mathit{arena}(A \uplus \{x \mapsto n\}) \tag{4.8}$$

The *mod_inv* predicate describes an arena, while the *token* predicate describes an arena that contains at x an allocated chunk of size n.

We soon encounter a problem. The postcondition for `malloc` separately-conjoins *mod_inv* and *token*, but these predicates do not describe separate parts of the heap. In fact, they describe exactly the *same* part of the heap. Is it necessary for both *mod_inv* and *token* to describe the whole arena? The *mod_inv* predicate must, because it forms the entire precondition for `malloc`, and so is the only way for that procedure to know that the arena it inherits is valid. The $\mathit{token}\ x\ n$ predicate also must describe the whole arena, in order to express that the chunk at x is genuinely in the arena; that is, reachable from the head of the linked list. (In fact, it suffices for the $\mathit{token}\ x\ n$ predicate to describe only the initial segment of this linked list, up to x. Nonetheless, since x may appear at an arbitrary position in the list, this observation does not change the fact that *token* and *mod_inv* have a non-trivial overlap.)

Using RGSep in a sequential context

To find definitions for $token$ and mod_inv that describe the same piece of the heap, but can still be $*$-conjoined, we turn to RGSep. In the original setting of RGSep, a region of the heap is distinguished as the 'shared' part; it can then be repeatedly described using the box modality. In our non-concurrent setting, we shall recast this region as the module state. Where RGSep refers to the 'local state', we shall henceforth use the term 'client state'.

Let us adapt (4.6), (4.7) and (4.8) by boxing the definientia like so:

$$uninit \stackrel{\text{def}}{=} \boxed{\begin{array}{|c|c|}\hline s & \\ \hline 0 & 0 \\ \hline \end{array}} \tag{4.9}$$

$$mod\_inv \stackrel{\text{def}}{=} \boxed{\exists A.\ arena\ A} \tag{4.10}$$

$$token\ x\ n \stackrel{\text{def}}{=} \boxed{\exists A.\ arena(A \uplus \{x \mapsto n\})} \tag{4.11}$$

With these definitions, assertions such as $mod\_inv * token\ x\ n$ become satisfiable. (In fact, that assertion is equivalent to $token\ x\ n$.)

We do not need the full RGSep system. In particular, because we are in a non-concurrent setting, the rely shall always be the empty relation. Perhaps surprisingly, we still require the guarantee. We shall therefore refer to our usage of RGSep as 'GSep'. The GSep judgement

$$C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)$$

is defined to be the same as the RGSep judgement

$$C\ \mathbf{sat}_{\mathsf{RGSep}}\ (P, \emptyset, G, Q)$$

and GSep inherits all of the RGSep proof rules except RG-PAR. We shall be handling pre- and postconditions that are rather large, so we shall sometimes tabulate the GSep specification (P, G, Q) as follows:

PRE	GUAR	POST
P	G	Q

Why is the guarantee necessary?

We shall now explain why the guarantee is necessary. In ordinary separation logic, the frame rule is sound because the extra $*$-conjunct must describe state that is disjoint from that which is affected by the command. But in RGSep, the $*$-operator does not enforce disjointness on the module state. Hence, if we use the ordinary frame rule from Fig. 2.5, we can construct an instance where the framed assertion refers to module state that is modified by the command:

$$\frac{\vdash \left\{\boxed{\mathtt{x} \mapsto 3}\right\}\ \mathtt{[x]} := 2\ \left\{\boxed{\mathtt{x} \mapsto 2}\right\}}{\vdash \left\{\boxed{\mathtt{x} \mapsto 3} * \boxed{\mathtt{x} \mapsto 3}\right\}\ \mathtt{[x]} := 2\ \left\{\boxed{\mathtt{x} \mapsto 2} * \boxed{\mathtt{x} \mapsto 3}\right\}}\ \textsc{Frame}$$

By (2.7), the deduced precondition is equivalent to $\boxed{\mathtt{x} \mapsto 3}$ and the postcondition is equivalent to $\boxed{\mathtt{x} \mapsto 2 \wedge \mathtt{x} \mapsto 3}$, which is equivalent to $false$. The deduction is invalid, and hence the ordinary frame rule is unsound in RGSep. We could achieve soundness by requiring, as an additional side-condition, that C does not invalidate F, but this would violate compositionality. Instead of examining the implementation of C, we augment its specification with a guarantee, G, which summarises all of the changes to the module state that can be performed by C during

its execution. The frame rule then checks that no state change in G invalidates F. The rule becomes:

$$\frac{\vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q) \qquad F \text{ stable under } G \qquad wr(C) \not\frown rd(F)}{\vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P * F, G, Q * F)}\ \textsc{GSep-Frame}$$

In our memory manager example, the role of G is to describe the effect on the module state of arbitrary calls to `malloc` and to `free`. We set

$$G \stackrel{\text{def}}{=} \bigcup_x \mathit{Malloc}(x) \cup \bigcup_x \mathit{Free}(x)$$

where

$$\mathit{Malloc}(x) \stackrel{\text{def}}{=} (\mathit{uninit} \wedge A = \emptyset) \vee \mathit{arena}\ A \rightsquigarrow \mathit{arena}(A \uplus \{x \mapsto n\}) \tag{4.12}$$

$$\mathit{Free}(x) \stackrel{\text{def}}{=} \mathit{arena}(A \uplus \{x \mapsto n\}) \rightsquigarrow \mathit{arena}\ A. \tag{4.13}$$

The need for stability

In order to apply the GSep-Frame rule, which is necessary for verifying clients such as the one in Fig. 4.1, we must be able to confirm that the pre- and postconditions of `malloc` and `free` are stable under G.

The need for stability can be understood at a more intuitive level than as a technical side-condition on a proof rule. Immediately after a call to, say, `x := malloc(2 × WORD)`, there will be at `x` a 'gap' in the arena; that is, a chunk marked as allocated. Later, `free(x)` will be called and that gap will be filled. But between these calls, there may be several *other* calls to `malloc` and to `free`. As a result of these calls, some chunks may be allocated, some deallocated, extra memory may be requested from the system, and consecutive free chunks may be coalesced. We must ensure that the 'gap' survives its 'journey' from the postcondition of `malloc` to the precondition of the corresponding `free`, which is to say, we must ensure that it is stable.

In fact, it is not the case that all of the pre- and postconditions in (4.1) are stable. For instance, the assertion $\mathit{token}\ x\ n$, which is defined in (4.10), can be invalidated by the action $\mathit{Free}(x)$, which is in G. By $\mathit{token}\ x\ n$ we assert that x is an allocated chunk, but the Free action can de-allocate it.

This failure of stability is merely an artefact of our overly crude guarantee relation. In fact, we should not worry whether $\mathit{token}\ x\ n$ is stable under $\mathit{Free}(x)$, because the only call that can perform this action is `free`(x), and once the token has reached this call, its purpose has been served, and it no longer needs to be stable. We would prefer, then, to say that $\mathit{token}\ x\ n$ must be stable under all actions except $\mathit{Free}(x)$. Vafeiadis and Parkinson [2007] deal with similar situations in their original, concurrent setting of RGSep by parameterising actions by thread identifiers, then requiring assertions only to be stable under actions parameterised by the identifiers of other threads. Our non-concurrent setting does not provide an analogue of thread identifiers, so we revert to a suggestion made by Vafeiadis [2007, §4.3.3]: the use of fractional permissions.

Let us redefine the token and $\mathit{chunk}_{\mathsf{a}}$ predicates (from (4.11) and (4.3) respectively) as follows.

$$\mathit{token}\ x\ n \stackrel{\text{def}}{=} \boxed{\exists A.\ \mathit{arena}(A \uplus \{x \mapsto n\})} * \tfrac{1}{2}\left|\frac{x-1}{(x+n)_{\blacksquare}}\right| \tag{4.14}$$

$$\mathit{chunk}_{\mathsf{a}}\ x\ y \stackrel{\text{def}}{=} x \mathrel{\dot{<}} y * \tfrac{1}{2}\left|\frac{x}{y_{\blacksquare}}\right| \tag{4.15}$$

A client obtaining $token\,x\,n$ from `malloc` now receives not only an assurance about the module state, but also partial ownership of one of the pointers from which the arena is constructed: half of the pointer preceding x, to be precise. The $chunk_{\mathsf{a}}$ predicate is updated accordingly, so that it encompasses only the other half of that pointer.

The effect of these new definitions is that $token\,x\,n$ is now stable under $Free(x)$, by virtue of the fact that this action cannot happen without yielding the token. To see this, consider an arbitrary state satisfying $token\,x\,n$. The client state will contain half of the pointer at $(x-1)$, and because x is an allocated chunk in the arena, the module state will contain the other half. Then suppose that the module state is transformed by the $Free(x)$ action. This would place the entire pointer in the module state, leading to the impossibility of having three half-pointers at $(x-1)$. Therefore, in accordance with Defn. 2.19, $token\,x\,n$ is vacuously stable under $Free(x)$, and in fact, all of the other actions in G too.

Hiding the guarantee

The guarantee contains details about the internal workings of the module. We would therefore prefer to keep it hidden from clients. This is difficult, because in order for a client to use the GSep-Frame rule, they must be able to confirm stability under G. How can they do this without knowing the value of G?

Our solution to this problem is to employ explicit stabilisation. In the previous chapter we used explicit stabilisation to shift the burden of checking stability from the library to its client. Now, we shall use it to shift stability checks from the client to the module. Here are our proposed specifications for `malloc` and `free` (which we shall abbreviate as $spec_{\mathsf{m}}$ and $spec_{\mathsf{f}}$):

$$\begin{array}{rll} \texttt{malloc(nb)} & \mathbf{sat}_{\mathsf{GSep}} & spec_{\mathsf{m}} \\ \texttt{free(ap)} & \mathbf{sat}_{\mathsf{GSep}} & spec_{\mathsf{f}} \end{array}$$

where

$$spec_{\mathsf{m}} \stackrel{\text{def}}{=} \begin{array}{|c|c|c|} \hline \text{PRE} & \text{GUAR} & \text{POST} \\ \hline \lceil mod\_inv \vee uninit \rceil_G & G & \lfloor mod\_inv \rfloor_G \\ & & * \lfloor token\,\texttt{ret}\ \lceil \texttt{nb/WORD} \rceil \rfloor_G \\ & & * \left|\begin{array}{l|l} \texttt{ret} & \texttt{ret} + \lceil \texttt{nb/WORD} \rceil \end{array}\right. \\ \hline \end{array} \tag{4.16}$$

and

$$spec_{\mathsf{f}} \stackrel{\text{def}}{=} \begin{array}{|c|c|c|} \hline \text{PRE} & \text{GUAR} & \text{POST} \\ \hline \lceil mod\_inv \rceil_G & G & \lfloor mod\_inv \rfloor_G \\ * \lceil token\,\texttt{ap}\,n \rceil_G & & \\ * \left|\begin{array}{l|l} \texttt{ap} & \texttt{ap} + n \end{array}\right. & & \\ \hline \end{array} \tag{4.17}$$

Compared to those specifications in (4.1), we have added the guarantee component, and explicit stabilisation brackets around the mod_inv and $token$ predicates. We shall explain in Sect. 4.2.4 how these specifications can be used to verify clients.

The directions of the stabilisations are key for understanding how the specifications work. Consider `free`'s postcondition, $\lfloor mod\_inv \rfloor_G$. As a result of the stabilisation, it is stronger than the original postcondition, mod_inv. Yet strengthening a postcondition is not, in general, sound. That we *have* been able to do so soundly implies that no strengthening has actually occurred; i.e., that mod_inv is already stable. The same argument applies in reverse to the

apparently-weakened precondition, and to the specification for `malloc`. We are using the stabilisation operators as certificates of stability, which clients can call upon in their own stability arguments without having to know the actual value of G. Hence, we can universally quantify over G when verifying the client, and thus hide its value.

This tactic is embodied by the following proof rule. We call it GSEP-DERIVPROG because later we shall show how it can be derived from more primitive rules.

$$\frac{\begin{array}{c}\Delta;\emptyset \vdash \texttt{atomic}\, C_1\ \mathbf{sat}_{\mathsf{GSep}}\ (P_1, G, Q_1)\ \ \cdots\ \ \Delta;\emptyset \vdash \texttt{atomic}\, C_n\ \mathbf{sat}_{\mathsf{GSep}}\ (P_n, G, Q_n) \\ \forall G.\, (\emptyset; (k_1 \mapsto (\bar{x}_1, P_1, G, Q_1), \ldots, k_n \mapsto (\bar{x}_n, P_n, G, Q_n)) \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)) \\ \mathrm{dom}(\Delta)\ \not\frown\ (apn(P) \cup apn(Q))\end{array}}{\emptyset \vdash \texttt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \texttt{in}\ C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}$$

On the first line of antecedants, we verify the procedure bodies, and on the second line, we verify the client C, given a specification for each procedure. The procedure bodies are wrapped in atomic blocks: this is to deal with the technical requirement that the module state must only be accessed from within a critical region. In each judgement in the antecedants, two components precede the turnstile: an abstract predicate dictionary and a procedure specification dictionary. When verifying the procedure bodies, we make available – in the abstract predicate dictionary Δ – the definitions of the abstract predicates that appear in the specifications of those procedures. When verifying the client, these definitions are unavailable: Δ has been replaced with $\emptyset$. Thus far, this description resembles the PROG-ABST rule from Sect. 2.4. A key departure in this new rule is the universal quantification over G. This forces the client verification to be carried out with respect to an arbitrary guarantee.

4.2 GSep

We describe GSep, which is a special case of the RGSep proof system in which the rely is constantly empty. GSep is used for verifying sequential modules, so there is no proof rule for parallel composition. The assertion language is the same as RGSep's, with the addition of abstract predicates (which are useful for obscuring the module state) and explicit stabilisation (which is useful for hiding the module's internal interference).

4.2.1 Assertions

The assertion language of GSep is as follows:

$$P ::= p \mid \boxed{p} \mid P * P \mid \alpha(e, \ldots, e) \mid \lfloor P \rfloor_G \mid \lceil P \rceil_G \mid P \wedge P \mid P \vee P \mid \exists x.\, P.$$

where G is a relation on $\mathsf{Store} \times \mathsf{Heap}$. The meanings of the new constructions are lifted to GSep in the straightforward manner depicted in Fig. 4.2. GSep assertions (P) contain ordinary separation logic assertions (p). These ordinary separation logic assertions shall not contain abstract predicates – for then we would suffer the complication of requiring predicate environments at both levels.

After applying explicit stabilisation to GSep, all of the properties detailed in Sect. 3.1.1 continue to hold. The following lemmas describe some additional GSep-specific properties. The first asserts that client-state assertions are vacuously stable, as they cannot describe the module state.

$$
\begin{array}{rcl}
[\![p]\!](\delta) & = & \{(s,h,H,i) \mid (s,h,i) \in [\![p]\!]_{\mathsf{SL}} \wedge \text{defined}\,(h \uplus H)\} \\
[\![\boxed{p}]\!](\delta) & = & \{(s,\emptyset,H,i) \mid (s,H,i) \in [\![p]\!]_{\mathsf{SL}}\} \\
[\![P * Q]\!](\delta) & = & \{(s,h \uplus h',H,i) \mid (s,h,H,i) \in [\![P]\!](\delta) \wedge (s,h',H,i) \in [\![Q]\!](\delta)\} \\
[\![\alpha(e_1,\ldots,e_n)]\!](\delta) & = & \{(s,h,H,i) \mid \delta(\alpha)([\![e_1]\!](s,i),\ldots,[\![e_n]\!](s,i))\} \\
[\![\lfloor P \rfloor_G]\!](\delta) & = & \left\{(s,h,H,i) \;\middle|\; \begin{array}{c} \forall H'.\,(\exists s'.\,((s,H),(s',H')) \in (G\backslash h)^*) \\ \Rightarrow (s,h,H',i) \in [\![P]\!](\delta) \end{array}\right\} \\
[\![\lceil P \rceil_G]\!](\delta) & = & \left\{(s,h,H,i) \;\middle|\; \begin{array}{c} \exists H'.\,(\exists s'.\,((s',H'),(s,H)) \in (G\backslash h)^*) \\ \wedge (s,h,H',i) \in [\![P]\!](\delta) \end{array}\right\} \\
[\![P \wedge Q]\!](\delta) & = & [\![P]\!](\delta) \cap [\![Q]\!](\delta) \\
[\![P \vee Q]\!](\delta) & = & [\![P]\!](\delta) \cup [\![Q]\!](\delta) \\
[\![\exists x.\,P]\!](\delta) & = & \{(s,h,H,i) \mid \exists v.\,(s,h,H,i[x \mapsto v]) \in [\![P]\!](\delta)\}
\end{array}
$$

Figure 4.2: Semantics of GSep assertions

Lemma 4.2 (Explicit stabilisation of client-state assertions)**.**

$$\lfloor p \rfloor_G \;\Leftrightarrow\; \lceil p \rceil_G \;\Leftrightarrow\; p$$

Second, we describe the distributivity of the stabilisation operators over the $*$-operator.

Lemma 4.3 (Explicit stabilisation of separately-conjoined assertions)**.**

$$\lfloor P \rfloor_G * \lfloor Q \rfloor_G \;\Rightarrow\; \lfloor P * Q \rfloor_G \qquad\qquad \lceil P * Q \rceil_G \;\Rightarrow\; \lceil P \rceil_G * \lceil Q \rceil_G$$

Both of the implications in Lem. 4.3 are strict. To see this for the first, set

$$
\begin{array}{rcl}
P & \stackrel{\text{def}}{=} & \big(\boxed{\mathtt{t} \mapsto 0} * \mathtt{x} \mapsto 0\big) \vee \big(\boxed{\mathtt{t} \mapsto 1} * \mathtt{y} \mapsto 0\big) \\
Q & \stackrel{\text{def}}{=} & \big(\boxed{\mathtt{t} \mapsto 0} * \mathtt{y} \mapsto 0\big) \vee \big(\boxed{\mathtt{t} \mapsto 1} * \mathtt{x} \mapsto 0\big) \\
G & \stackrel{\text{def}}{=} & \mathtt{t} \mapsto 0 \rightsquigarrow \mathtt{t} \mapsto 1
\end{array}
$$

and, after evaluating the explicit stabilisations, observe that

$$\boxed{\mathtt{t} \mapsto 1} * \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0 \;\not\Leftarrow\; \boxed{\mathtt{t} \mapsto 0 \vee \mathtt{t} \mapsto 1} * \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto 0.$$

For the second, set

$$
\begin{array}{rcl}
P & \stackrel{\text{def}}{=} & \boxed{\exists n.\,\mathtt{t} \mapsto n \wedge n < 0} \\
Q & \stackrel{\text{def}}{=} & \boxed{\exists n.\,\mathtt{t} \mapsto n \wedge n > 0} \\
G & \stackrel{\text{def}}{=} & \mathtt{t} \mapsto n \rightsquigarrow \mathtt{t} \mapsto n+1
\end{array}
$$

and, after evaluating the explicit stabilisations, observe that

$$\mathit{false} \;\not\Leftarrow\; \boxed{\exists n.\,\mathtt{t} \mapsto n \wedge n > 0}.$$

A bidirectional version of Lem. 4.3 is as follows.

Lemma 4.4.

$$\lfloor P \rfloor_G * \lfloor Q \rfloor_G \;\Leftrightarrow\; \lfloor \lfloor P \rfloor_G * \lfloor Q \rfloor_G \rfloor_G \qquad\qquad \lceil \lceil P \rceil_G * \lceil Q \rceil_G \rceil_G \;\Leftrightarrow\; \lceil P \rceil_G * \lceil Q \rceil_G$$

4.2.2 Judgements

Judgements in GSep are of the form

$$\Delta; \Gamma \vdash C \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q)$$

for commands, or

$$\Delta \vdash \mathit{prog} \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q)$$

for complete programs, where Δ is an abstract predicate dictionary and Γ is a procedure specification dictionary. The meaning of such judgements is given below; essentially, they are a straightforward amalgamation of RGSep judgements (Defn. 2.21) with abstract predicates and procedures (Defn. 2.5). Note also that we use a fractional model of the heap in order to support predicate definitions such as (4.14), and hence we use the $+$-operator from (2.4) rather than the $\uplus$-operator to combine heaps.

Definition 4.5 (Meaning of judgements: GSep)**.** Let $\mathit{safe}_{\mathsf{GSep}}(Q, G, \eta, \delta)$ be the largest set containing only those quintuples (C, s, h, H, i) that:

- do not fault (even in the presence of a module heap H and extra heap locations h_o:

$$\forall h_o, h_1.\, h + H + h_o = h_1 \implies (C, (s, h_1)) \not\rightarrow_\eta \mathbf{abort},$$

- satisfy the postcondition Q if they are terminal:

$$C = \mathtt{skip} \implies (s, h, H, i) \in [\![Q]\!](\delta),$$

- and continue to satisfy these properties after any execution step (whose associated module-state transition must be in the guarantee G), noting that this execution step may occur in the presence of extra heap locations h_o that are unaffected by the step:

$$\begin{array}{l}\forall h_1, h_o, C', s', h'.\\ \quad h + H + h_o = h_1 \wedge (C, (s, h_1)) \rightarrow_\eta (C', (s', h_1')) \\ \quad \implies (\exists h', H'.\, h_1' = h' + H' + h_o \wedge ((s, H), (s', H')) \in G \\ \qquad\qquad \wedge\ (C', s', h', H', i) \in \mathit{safe}_{\mathsf{GSep}}(Q, G, \eta, \delta)).\end{array}$$

We can then define:

$$\begin{array}{rcl} \models_{\eta,\delta} C \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q) & \stackrel{\text{def}}{=} & \forall (s, h, H, i) \in [\![P]\!](\delta). \\ & & (C, s, h, H, i) \in \mathit{safe}_{\mathsf{GSep}}(Q, G, \eta, \delta) \\ \eta \ \mathsf{implements}_\delta^{\mathsf{GSep}} \ \Gamma & \stackrel{\text{def}}{=} & \forall (k \mapsto (\bar{x}, P, G, Q)) \in \Gamma.\, \exists C. \\ & & \eta(k) = (\bar{x}, C) \wedge \models_{\eta,\delta} C \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q) \\ \Delta; \Gamma \models C \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q) & \stackrel{\text{def}}{=} & \forall \delta \in [\![\Delta]\!].\, \forall \eta. \\ & & \eta \ \mathsf{implements}_\delta^{\mathsf{GSep}} \ \Gamma \implies \models_{\eta,\delta} C \ \mathbf{sat}_{\mathsf{GSep}} \ (P, G, Q) \end{array}$$

Definition 4.6 (Meaning of judgements: GSep, for complete programs)**.** Let $\mathit{safe}_{\mathsf{GSep}}(Q, G, \delta)$ be the largest set containing only those quintuples $(\mathit{prog}, s, h, H, i)$ that:

$$\textsc{GSep-Atomic}\ \frac{\left\{p * p'\right\}\ C\ \left\{q * q'\right\} \qquad [\![p \rightsquigarrow q]\!] \subseteq G \qquad p, q \text{ precise}}{\Delta; \emptyset \vdash \texttt{atomic}\, C\ \mathbf{sat}_{\mathsf{GSep}}\ (\boxed{p} * p', G, \boxed{q} * q')}$$

$$\textsc{GSep-Frame}\ \frac{\Delta; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q) \qquad wr(C) \not\frown rd(F)}{\Delta; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P * F, G, Q * \lceil F \rceil_G)}$$

$$\textsc{GSep-Conseq}\ \frac{\Delta \models P \Rightarrow P' \qquad \Delta \models Q' \Rightarrow Q \qquad \Delta; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P', G, Q')}{\Delta; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}$$

$$\textsc{GSep-PredI}\ \frac{\Delta; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}{\Delta \uplus \Delta'; \Gamma \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}$$

$$\textsc{GSep-PredE}\ \frac{\Delta \uplus \Delta' \vdash \mathit{prog}\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q) \qquad \mathrm{dom}(\Delta') \not\frown (apn(P) \cup apn(Q) \cup apn(\Delta))}{\Delta \vdash \mathit{prog}\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}$$

$$\textsc{GSep-Prog}\ \frac{\Delta; \emptyset \vdash \texttt{atomic}\, C_1\ \mathbf{sat}_{\mathsf{GSep}}\ (P_1, G, Q_1)\ \cdots\ \Delta; \emptyset \vdash \texttt{atomic}\, C_n\ \mathbf{sat}_{\mathsf{GSep}}\ (P_n, G, Q_n) \qquad \Delta; (k_1 \mapsto (\bar{x}_1, P_1, G, Q_1), \ldots, k_n \mapsto (\bar{x}_n, P_n, G, Q_n)) \vdash C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}{\Delta \vdash \texttt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \texttt{in}\ C\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)}$$

Figure 4.3: Proof rules for GSep

- do not fault (even in the presence of a module heap H and extra heap locations h_o:

$$\forall h_o, h_1.\, h + H + h_o = h_1 \Longrightarrow (\mathit{prog}, (s, h_1)) \not\rightarrow_\eta \mathbf{abort},$$

- satisfy the postcondition Q if they are terminal:

$$(\exists \eta.\, \mathit{prog} = \texttt{procs}\ \eta\ \texttt{in}\ \texttt{skip}) \Longrightarrow (s, h, H, i) \in [\![Q]\!](\delta),$$

- and continue to satisfy these properties after any execution step (whose associated module-state transition must be in the guarantee G), noting that this execution step may occur in the presence of extra heap locations h_o that are unaffected by the step:

$$\begin{aligned}&\forall h_1, h_o, \mathit{prog}', s', h'.\\ &\quad h + H + h_o = h_1 \wedge (\mathit{prog}, (s, h_1)) \rightarrow (\mathit{prog}', (s', h_1'))\\ &\quad \Longrightarrow (\exists h', H'.\, h_1' = h' + H' + h_o \wedge ((s, H), (s', H')) \in G\\ &\qquad\qquad \wedge (\mathit{prog}', s', h', H', i) \in \mathit{safe}_{\mathsf{GSep}}(Q, G, \delta)).\end{aligned}$$

We can then define:

$$\begin{aligned}\Delta \models \mathit{prog}\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q) \stackrel{\text{def}}{=}\ &\forall \delta \in [\![\Delta]\!].\, \forall (s, h, H, i) \in [\![P]\!](\delta).\\ &(\mathit{prog}, s, h, H, i) \in \mathit{safe}_{\mathsf{GSep}}(Q, G, \delta)\end{aligned}$$

4.2.3 Proof rules

Figure 4.3 presents some proof rules for GSep.

The GSEP-PREDI, GSEP-PREDE, GSEP-CONSEQ and GSEP-PROG rules come straight out of the proof system for abstract predicates, as presented in Sect. 2.4. The GSEP-FRAME rule is similar to the RGSEP-FRAME rule, except that it uses explicit stabilisation in the post-condition rather than a stability check. The GSEP-ATOMIC rule is a descendant of the RGSEP-ATOMIC rule, but is considerably simpler, firstly because there is no longer a rely under which stability must be checked, and secondly because the body of the atomic block gains access to the *entirety* of the module state, rather than just part of it, as in the RGSep rule.

The GSEP-DERIVPROG rule presented in Sect. 4.1.3 can be derived with the help of the rules in Fig. 4.3, as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall G.\,(\emptyset; (k_i \mapsto (\bar{x}_i, P_i, G, Q_i))_{i\in n} \vdash C \textbf{ sat } (P,G,Q))}{\emptyset; (k_i \mapsto (\bar{x}_i, P_i, G, Q_i))_{i\in n} \vdash C \textbf{ sat } (P,G,Q)}\ \forall\text{-elimination}}{\Delta; (k_i \mapsto (\bar{x}_i, P_i, G, Q_i))_{i\in n} \vdash C \textbf{ sat } (P,G,Q)}\ \textsc{GSep-PredI}}{\Delta \vdash \texttt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \texttt{in}\ C \textbf{ sat}_{\mathsf{GSep}}\ (P,G,Q)}\ \textsc{GSep-Prog}}{\emptyset \vdash \texttt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \texttt{in}\ C \textbf{ sat}_{\mathsf{GSep}}\ (P,G,Q)}\ \textsc{GSep-PredE}$$

(†) $\Delta; (k_i \mapsto (\bar{x}_i, P_i, G, Q_i))_{i\in n} \vdash C \textbf{ sat } (P,G,Q)$

(*) $\Delta \vdash \texttt{procs}\ k_1(\bar{x}_1) \stackrel{\text{def}}{=} C_1, \ldots, k_n(\bar{x}_n) \stackrel{\text{def}}{=} C_n\ \texttt{in}\ C \textbf{ sat}_{\mathsf{GSep}}\ (P,G,Q)$

where (*) stands for

$$\mathrm{dom}(\Delta) \not\cap (apn(P) \cup apn(Q))$$

and (†) stands for

$$(\Delta; \emptyset \vdash \texttt{atomic}\, C_i \textbf{ sat } (P_i, G, Q_i))_{i\in n}.$$

4.2.4 Application to the memory manager

Let us look at how the GSEP-DERIVPROG rule is used to verify the combined system of the memory manager and a client C. Let C_{m} and C_{f} stand for the implementations of `malloc` and `free` respectively, and let $spec_{\mathsf{m}}$ and $spec_{\mathsf{f}}$ be their specifications, defined like so:

$spec_{\mathsf{m}} \stackrel{\text{def}}{=}$

PRE	GUAR	POST
$\lceil mod\_inv \rceil_G$	G	$\lfloor mod\_inv \rfloor_G * \lfloor token\ \texttt{ret}\ \lceil \frac{\texttt{nb}}{\texttt{WORD}} \rceil \rfloor_G * \left\lvert \begin{smallmatrix} \texttt{ret} \\ \\ \end{smallmatrix} \right\rvert^{\texttt{ret} + \lceil \texttt{nb/WORD} \rceil}$

$spec_{\mathsf{f}} \stackrel{\text{def}}{=}$

PRE	GUAR	POST
$\lceil mod\_inv \rceil_G * \lceil token\ \texttt{ap}\ n \rceil_G * \left\lvert \begin{smallmatrix} \texttt{ap} \\ \\ \end{smallmatrix} \right\rvert^{\texttt{ap}+n}$	G	$\lfloor mod\_inv \rfloor_G$

Let $spec$ be the specification of the client. For this, we use a precondition that describes an uninitialised arena, and take $true$ as the final postcondition.

$spec \stackrel{\text{def}}{=}$

PRE	GUAR	POST
$uninit$	G	$true$

Then:

$$\dfrac{\begin{array}{c}\{token \mapsto \ldots, mod\_inv \mapsto \ldots\}; \emptyset \vdash \texttt{atomic}\, C_{\mathsf{m}} \textbf{ sat}_{\mathsf{GSep}}\ spec_{\mathsf{m}} \\ \{token \mapsto \ldots, mod\_inv \mapsto \ldots\}; \emptyset \vdash \texttt{atomic}\, C_{\mathsf{f}} \textbf{ sat}_{\mathsf{GSep}}\ spec_{\mathsf{f}} \\ \forall G.\left(\emptyset; \left\{\begin{array}{l}\texttt{malloc} \mapsto (\texttt{nb}, spec_{\mathsf{m}}), \\ \texttt{free} \mapsto (\texttt{ap}, spec_{\mathsf{f}})\end{array}\right\} \vdash C \textbf{ sat}_{\mathsf{GSep}}\ spec\right)\end{array}}{\emptyset \vdash \texttt{procs}\ \texttt{malloc}(\texttt{nb}) \stackrel{\text{def}}{=} C_{\mathsf{m}}, \texttt{free}(\texttt{ap}) \stackrel{\text{def}}{=} C_{\mathsf{f}}\ \texttt{in}\ C \textbf{ sat}_{\mathsf{GSep}}\ spec}\ \textsc{GSep-DerivProg}$$

$\{uninit\}$
$\{\lceil mod\_inv \vee uninit\rceil_G\}$
`u := malloc(2 * WORD);`
$\{\lceil mod\_inv\rceil_G * \lceil token\,\mathtt{u}\,2\rceil_G * \boxed{\mathtt{u}}\}$
`v := malloc(3 * WORD);`
$\{\lceil mod\_inv\rceil_G * \lceil token\,\mathtt{u}\,2\rceil_G * \boxed{\mathtt{u}} * \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}}\}$
`[u+1]:=5;`
$\{\lceil mod\_inv\rceil_G * \lceil token\,\mathtt{u}\,2\rceil_G * \boxed{\mathtt{u}}_{5} * \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}}\}$
`free(u);`
$\{\lceil mod\_inv\rceil_G * \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}}\}$
`free(v);`
$\{\lfloor mod\_inv\rfloor_G\}$
$\{true\}$

Figure 4.4: Proof outline of a simple client using the specifications in (4.16) and (4.17)

Observe that the definitions of the $token$ and mod_inv predicates, and the value of G, are duly hidden from the client verification.

Figure 4.4 shows the specifications from (4.16) and (4.17) in action, being used to verify a simple client of the memory manager. Let us focus on the '`free`(`u`)' step. From the specification for `free` we have

$$\vdash \mathtt{free(u)}\ \mathbf{sat}_{\mathsf{GSep}}\ (P, G, Q)$$

where

$$\begin{array}{rcl} P & = & \lceil mod\_inv\rceil_G * \lceil token\,\mathtt{u}\,2\rceil_G * \boxed{\mathtt{u}}_{5} \\ Q & = & \lceil mod\_inv\rceil_G\,. \end{array}$$

Via the GSep-Frame rule, we deduce that `free`(`u`) satisfies the following GSep specification

PRE	GUAR	POST
$P * \left\lceil \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}} \right\rceil_G$	G	$Q * \left\lceil \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}} \right\rceil_G$

which, by Lem. 4.2, is equivalent to

PRE	GUAR	POST
$P * \left\lceil \lceil token\,\mathtt{v}\,3\rceil_G * \left\lceil \boxed{\mathtt{v}} \right\rceil_G \right\rceil_G$	G	$Q * \left\lceil \lceil token\,\mathtt{v}\,3\rceil_G * \left\lceil \boxed{\mathtt{v}} \right\rceil_G \right\rceil_G$

and hence, by Lem. 4.4, to

PRE	GUAR	POST
$P * \lceil token\,\mathtt{v}\,3\rceil_G * \left\lceil \boxed{\mathtt{v}} \right\rceil_G$	G	$Q * \lceil token\,\mathtt{v}\,3\rceil_G * \left\lceil \boxed{\mathtt{v}} \right\rceil_G$

and hence, by Lem. 4.2 again, to

PRE	GUAR	POST
$P * \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}}$	G	$Q * \lceil token\,\mathtt{v}\,3\rceil_G * \boxed{\mathtt{v}}$

as required.

4.3 Details of the verification

Before presenting the detailed proof of the implementation of the Version 7 Unix memory manager, we discuss a few necessary refinements to the story that has been developed so far.

4.3.1 Failure to allocate

So far, we have assumed that every call to `malloc` results in a chunk of the requested size being returned, but in fact, `malloc` may fail. When it does so, it returns the null pointer. Accordingly, we update the specification given in (4.16) to include an extra disjunct that handles this case.

	PRE	GUAR	POST
$\texttt{malloc(nb)}\ \mathbf{sat}_{\mathsf{GSep}}$	$\lceil mod\_inv \vee uninit \rceil_G$	G	$\lfloor mod\_inv \rfloor_G * \left(\begin{array}{l} \left(\begin{array}{l} \lfloor token\, \texttt{ret}\ \lceil \frac{\texttt{nb}}{\texttt{WORD}} \rceil \rfloor_G \\ * \left\lvert \texttt{ret} \right\rvert\!\!\left\lvert \texttt{ret} + \lceil \texttt{nb/WORD} \rceil \right. \end{array} \right) \\ \vee\, \texttt{ret} \doteq 0 \end{array} \right)$

We also add to the guarantee G a new action, called Tau, that represents the case where `malloc` runs, but does not modify the set of allocated chunks. This action is named after the 'internal action' in process calculi.

$$Tau \stackrel{\text{def}}{=} (uninit \wedge A = \emptyset) \vee arena\, A \ \rightsquigarrow\ arena\, A \tag{4.18}$$

4.3.2 Extending the arena

If no suitable chunk can be found in the arena, `malloc` invokes `sbrk`. This system routine seeks to increase the size of the heap available to the process by at least `n` bytes.

$$\left\{ brka(x) \right\}\ \texttt{sbrk(n)}\ \left\{ \begin{array}{l} (brka(x) * \texttt{ret} \doteq \frac{-1}{\texttt{WORD}} * \texttt{n} \dot{\neq} 0) \vee \\ (brka(\texttt{ret} + \lceil \frac{\texttt{n}}{\texttt{WORD}} \rceil) * x \dot{\leq} \texttt{ret} * \left\lvert \texttt{ret} \right\rvert\!\!\left\lvert \texttt{ret} + \lceil \texttt{n/WORD} \rceil \right.) \end{array} \right\}$$

In the specification above, the $brka(x)$ predicate means that the breakpoint (the first address outside the currently available portion of memory) is at or after x. A successful call to `sbrk` advances this breakpoint. The $brka$ predicate places no upper bound on the value of the breakpoint; we find that this makes the specification more natural to use. The return value is -1 in the case of failure, and the address of the first newly-available cell in the case of success. A call to $\texttt{sbrk}(0)$ will always succeed.

We update the definitions of the $arena$ predicate from (4.5), and of the $uninit$ predicate from (4.9), to include descriptions of the breakpoint.

$$arena\, A \stackrel{\text{def}}{=} \exists C.\, chunks\, \texttt{s}\, \texttt{t}\, C * A \doteq C^{\texttt{a}} * \left\lvert \frac{\texttt{t}}{\texttt{s}_{\blacksquare}} \right\rvert * brka(\texttt{t}+1) \tag{4.19}$$

$$uninit \stackrel{\text{def}}{=} \boxed{\left\lvert \frac{s}{0} \right\rvert\!\!\left\lvert \frac{}{0} \right\rvert * brka(\texttt{s}+2)} \tag{4.20}$$

4.3.3 Gaps in the arena

When new memory is obtained from the system via a call to `sbrk`, the start of the newly-available memory may not immediately follow the end of the previously-available memory. There may be a gap, resulting from another process making a call to `sbrk`.

The Version 7 Unix memory manager is able, nonetheless, to maintain the appearance of a contiguous arena, by simply marking this gap as an allocated chunk, never to be freed. This policy slightly complicates our verification, because an allocated chunk is defined to be one whose pointer is partially owned by the client to whom it is allocated. Chunks that represent a gap in the arena are not allocated to any client, so there is nowhere for the other half of the pointer to go.

Our solution to this is to introduce a third kind of chunk, which we shall call a 'system chunk'. The busy status τ shall now range over the set $\{\mathtt{u}, \mathtt{a}, \mathtt{s}\}$. A system chunk is the same as an allocated chunk, but the entire pointer is stored in the arena, rather than just half of it.

$$chunk_{\mathtt{s}}\, x\, y \stackrel{\text{def}}{=} x \dot{<} y * \left|\frac{x}{y_{\blacksquare}}\right| \tag{4.21}$$

4.3.4 The designated victim

The `malloc` routine does not always begin its search for a suitable chunk at the start of the arena, but rather, at the chunk at position `v`. Being the most likely to be picked, this chunk is known as the designated victim. After a call to `free`, the designated victim is the just-freed chunk. After a call to `malloc`, the designated victim is the chunk immediately following the just-allocated chunk.

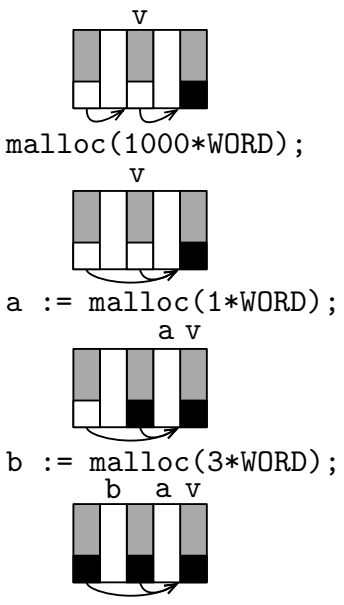
We accommodate the designated victim by redefining our *arena* predicate from (4.19) like so.

$$\begin{aligned} arena\, A \stackrel{\text{def}}{=}\ & \exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \\ & A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * brka(\mathtt{t}+1) \end{aligned} \tag{4.22}$$

Essentially, we have split the list of chunks into two lists, C_1 and C_2; these respectively describe the sections of the arena before and after `v`.

This feature of the memory manager's implementation harbours a subtle bug, discovered by Mike Dodds. The implementation does not update `v` if allocation fails, but it should, for the following reason. It is possible that, before a call to `malloc`, `v` points to the second of two unallocated chunks. During the (unsuccessful) call, those two chunks may be coalesced, leaving `v` pointing *inside* a chunk.

The picture to the right demonstrates how this bug could wreak havoc. Our contrived arena contains just two one-word chunks, both of which are free, and `v` initially points to the second. The first `malloc` call fails, but has the side-effect of leaving `v` inside the coalesced chunk. We then allocate a small chunk at `a`, before wrapping around to the start of the arena and allocating a larger chunk at `b`, thereby reaching a situation in which the contents of the smaller chunk is allocated twice.



The discovery of this bug was prompted by the failure of the invariant $chunks\ \mathtt{s}\ \mathtt{v}\ C_1$, which states that `v` identifies a valid pointer in the arena. We have successfully executed our exploit to confirm that the bug is real.

4.3.5 Program variables as predicate parameters

Formally, any program variable that appears in a predicate definition must be given as a parameter to that predicate. For instance, the $arena$ predicate from (4.22) should have `s`, `v` and `t` passed as parameters, like so.

$$\begin{aligned} arena\, A\, s\, v\, t \stackrel{\text{def}}{=} \exists C_1, C_2.\ & chunks\, s\, v\, C_1 * chunks\, v\, t\, C_2 * \\ & A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{t}{s_{\blacksquare}}\right| * brka(t+1) \end{aligned}$$

We now write $arena\, A\, \mathtt{s}\, \mathtt{v}\, \mathtt{t}$ to describe our arena. This practice ensures that when the $arena$ predicate appears in an assertion, it is clear which program variables are involved. This is necessary when, for instance, using the frame rule, which requires that the framed assertion does not involve any program variables modified by the command. The mod_inv, $uninit$ and $token$ predicates are parameterised in a similar way.

4.3.6 Collected definitions

Having revised the definitions of many of our predicates several times, the following table summarises the final version of each predicate.

Types

$$\begin{aligned}
\mathsf{tag} &\stackrel{\text{def}}{=} \{\mathsf{u}, \mathsf{a}, \mathsf{s}\} \qquad \text{(unallocated/allocated/system)} \\
\mathsf{ptr} &\stackrel{\text{def}}{=} \{x \in \mathbb{R} \mid x \times \texttt{WORD} \in \mathbb{Z}\} \\
\mathsf{chunks} &\stackrel{\text{def}}{=} \{C : (\mathbb{N}^+ \times \mathsf{tag} \times \mathbb{N}^+)\ \mathsf{list} \mid \\
&\qquad \text{each } \langle x, \tau, y\rangle \text{ in } C \text{ satisfies } x < y, \text{ and each} \\
&\qquad \text{consecutive pair } \langle x, \tau, y\rangle, \langle x', \tau', y'\rangle \text{ satisfies } y \le x'\} \\
\mathsf{achunks} &\stackrel{\text{def}}{=} \{A : \mathbb{N}^+ \rightharpoonup_{\text{fin}} \mathbb{N} \mid \\
&\qquad \forall x, y \in \mathrm{dom}(A).\, x < y \Longrightarrow x + A(x) \le y\}
\end{aligned}$$

Operators

$$\begin{aligned}
(-)^{\mathsf{a}} : \mathsf{chunks} \to \mathsf{achunks} &\stackrel{\text{def}}{=} \lambda C.\, \{(x+1 \mapsto y-x-1) \mid \langle x, \mathsf{a}, y\rangle \in C\} \\
(-)_{\blacksquare} : \mathsf{ptr} \to \mathsf{ptr} &\stackrel{\text{def}}{=} \lambda y.\, y + \tfrac{1}{\texttt{WORD}} \\
(A : \mathsf{achunks}) \uplus (A' : \mathsf{achunks}) &\stackrel{\text{def}}{=} \begin{cases} A \cup A' & \text{if } \mathrm{dom}(A) \not\between \mathrm{dom}(A') \\ \text{undefined} & \text{otherwise} \end{cases} \\
(C : \mathsf{chunks}) \circ (C' : \mathsf{chunks}) &\stackrel{\text{def}}{=} \begin{cases} \text{concatenation of } C \text{ and } C' & \text{if result is a valid element of chunks} \\ \text{undefined} & \text{otherwise} \end{cases} \\
(C : \mathsf{chunks}) \circ\!\!- (C' : \mathsf{chunks}) &\stackrel{\text{def}}{=} \begin{cases} C'' & \text{if } C = C' \circ C'' \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Predicates

$$\begin{aligned}
\mathit{chunk}_{\mathsf{u}}(x : \mathsf{ptr})\,(y : \mathsf{ptr}) &\stackrel{\text{def}}{=} \left|\begin{array}{c|c} x & \\ \hline y & \end{array}\right|^{y} \\
\mathit{chunk}_{\mathsf{a}}(x : \mathsf{ptr})\,(y : \mathsf{ptr}) &\stackrel{\text{def}}{=} x \doteq y * \tfrac{1}{2}\left|\begin{array}{c|} x \\ \hline y_{\blacksquare} \end{array}\right| \\
\mathit{chunk}_{\mathsf{s}}(x : \mathsf{ptr})\,(y : \mathsf{ptr}) &\stackrel{\text{def}}{=} x \doteq y * \left|\begin{array}{c|} x \\ \hline y_{\blacksquare} \end{array}\right| \\
\mathit{chunks}(x : \mathsf{ptr})\,(y : \mathsf{ptr})\,(C : \mathsf{chunks}) &\stackrel{\text{def}}{=} (x \doteq y * C \doteq [\,]) \vee \exists z : \mathsf{ptr}.\, \exists \tau : \mathsf{tag}. \\
&\qquad \mathit{chunk}_{\tau}\ x\, z * \mathit{chunks}\ z\, y\, (C \circ\!\!- [\langle x, \tau, z\rangle]) \\
\mathit{arena}\,(s : \mathsf{ptr})\,(v : \mathsf{ptr}) &\stackrel{\text{def}}{=} \exists C_1, C_2 : \mathsf{chunks}. \\
(t : \mathsf{ptr})\,(A : \mathsf{achunks}) &\qquad \mathit{chunks}\ s\, v\, C_1 * \mathit{chunks}\ v\, t\, C_2 \\
&\qquad * A \doteq (C_1 \circ C_2)^{\mathsf{a}} * \left|\begin{array}{c|} t \\ \hline s_{\blacksquare} \end{array}\right| * \mathit{brka}(t+1) \\
\mathit{mod_inv}\,(s : \mathsf{ptr})\,(v : \mathsf{ptr})\,(t : \mathsf{ptr}) &\stackrel{\text{def}}{=} \boxed{\exists A : \mathsf{achunks}.\ \mathit{arena}\ s\, v\, t\, A} \\
\mathit{uninit}\,(s : \mathsf{ptr}) &\stackrel{\text{def}}{=} \boxed{\left|\begin{array}{c|c} s & \\ \hline 0 & 0 \end{array}\right| * \mathit{brka}(s+2)} \\
\mathit{token}\,(s : \mathsf{ptr})\,(v : \mathsf{ptr})\,(t : \mathsf{ptr}) &\stackrel{\text{def}}{=} \boxed{\begin{array}{l} \exists A : \mathsf{achunks}. \\ \mathit{arena}\ s\, v\, t\, (A \uplus \{x \mapsto n\}) \end{array}} * \tfrac{1}{2}\left|\begin{array}{c|} x-1 \\ \hline (x+n)_{\blacksquare} \end{array}\right| \\
(x : \mathsf{ptr})\,(n : \mathbb{N}^+) &
\end{aligned}$$

Actions and guarantees

$$
\begin{aligned}
\mathit{Malloc}(x) &\stackrel{\text{def}}{=} (\mathit{uninit}\,\mathtt{s} \wedge A = \emptyset) \vee \mathit{arena}\ \mathtt{s\,v\,t}\ A \ \rightsquigarrow\ \mathit{arena}\,\mathtt{s\,v\,t}\,(A \uplus \{x \mapsto n\}) \\
\mathit{Free}(x) &\stackrel{\text{def}}{=} \mathit{arena}\,\mathtt{s\,v\,t}\,(A \uplus \{x \mapsto n\}) \ \rightsquigarrow\ \mathit{arena}\,\mathtt{s\,v\,t}\,A \\
\mathit{Tau} &\stackrel{\text{def}}{=} (\mathit{uninit}\,\mathtt{s} \wedge A = \emptyset) \vee \mathit{arena}\,\mathtt{s\,v\,t}\,A \ \rightsquigarrow\ \mathit{arena}\,\mathtt{s\,v\,t}\,A \\
G &\stackrel{\text{def}}{=} \textstyle\bigcup_x \mathit{Malloc}(x) \ \cup\ \bigcup_x \mathit{Free}(x) \ \cup\ \mathit{Tau}
\end{aligned}
$$

Specifications

$$
\left\{\mathit{brka}(x)\right\}\ \mathtt{sbrk(n)}\ \left\{\begin{array}{l}(\mathit{brka}(x) * \mathtt{ret} \doteq \frac{-1}{\mathtt{WORD}} * \mathtt{n} \neq 0) \vee \\ (\mathit{brka}(\mathtt{ret} + \lceil \frac{\mathtt{n}}{\mathtt{WORD}} \rceil) * x \stackrel{\cdot}{\leq} \mathtt{ret} * \left|\!\!\begin{array}{|c}\hline\mathtt{ret}\\ \hline\end{array}\!\!\right|^{\mathtt{ret} + \lceil \mathtt{n/WORD} \rceil})\end{array}\right\}
$$

	PRE	GUAR	POST
$\mathtt{malloc(nb)}\ \mathbf{sat}_{\mathsf{GSep}}$	$\left\lceil \begin{array}{l}\mathit{mod\_inv}\,\mathtt{s\,v\,t} \\ \vee\ \mathit{uninit}\,\mathtt{s}\end{array} \right\rceil_G$	G	$\left\lfloor \mathit{mod\_inv}\,\mathtt{s\,v\,t} \right\rfloor_G * \left(\begin{array}{l}\left(\begin{array}{l}\left\lfloor \mathit{token}\,\mathtt{s\,v\,t\,ret}\ \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil \right\rfloor_G \\ * \left|\!\!\begin{array}{|c}\hline\mathtt{ret}\\ \hline\end{array}\!\!\right|^{\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil}\end{array}\right) \\ \vee\ \mathtt{ret} \doteq 0\end{array}\right)$

	PRE	GUAR	POST
$\mathtt{free(ap)}\ \mathbf{sat}_{\mathsf{GSep}}$	$\begin{array}{l}\lceil \mathit{mod\_inv}\,\mathtt{s\,v\,t} \rceil_G \\ * \lceil \mathit{token}\,\mathtt{s\,v\,t\,ap}\,n \rceil_G \\ * \left|\!\!\begin{array}{|c}\hline\mathtt{ap}\\ \hline\end{array}\!\!\right|^{\mathtt{ap}+n}\end{array}$	G	$\left\lfloor \mathit{mod\_inv}\,\mathtt{s\,v\,t} \right\rfloor_G$

The following lemmas state some useful consequences of the definitions given above, and are used in the upcoming proof.

Lemma 4.7. *Two consecutive lists of chunks can be concatenated.*

$$\mathit{chunks}\ x\ y\ C_1 * \mathit{chunks}\ y\ z\ C_2 \implies \mathit{chunks}\ x\ z\ (C_1 \circ C_2)$$

Lemma 4.8. *A single chunk is a special case of a list of chunks.*

$$\mathit{chunk}_\tau\ x\ y \implies \mathit{chunks}\ x\ y\ [\langle x, \tau, y\rangle]$$

Lemma 4.9. *Two lists of chunks, C_1 and C_2, where C_1 precedes C_2 without overlap, can be legally concatenated.*

$$\mathit{chunks}\ w\ x\ C_1 * x \stackrel{\cdot}{\leq} y * \mathit{chunks}\ y\ z\ C_2 \implies (C_1 \circ C_2)\ \textit{is defined}$$

Lemma 4.10. *In any valid arena, the cell at s is allocated.*

$$\mathit{arena}\ s\ v\ t\ A \implies \exists n > 0.\ \left|\tfrac{s}{n}\right| * (\left|\tfrac{s}{n}\right| \mathrel{-\!\!*} \mathit{arena}\ s\ v\ t\ A)$$

Lemma 4.11. *A list of chunks can be split at any intermediate chunk.*

$$\mathit{chunks}\ w\ x\ C \wedge \langle y, \tau, z\rangle \in C \implies \exists C_1, C_2.\ \mathit{chunks}\ w\ y\ C_1 * \mathit{chunk}_\tau\ y\ z * \mathit{chunks}\ z\ x\ C_2$$

Lemma 4.12. *The brka predicate imposes only a lower bound on the breakpoint.*

$$x \leq y \wedge \mathit{brka}(y) \implies \mathit{brka}(x)$$

4.3.7 Mutating program variables

Even in the final versions given in the previous subsection, we have been somewhat disingenuous about the program variables `t` and `v`, which mark the end of the arena and the designated victim, respectively. Either may change during the execution of the memory manager's procedures: `t` is advanced after a successful call to `sbrk`, and assignments to `v` occur in both `malloc` and `free`. Yet in RGSep, and hence in GSep too, interference is only permitted to happen via the heap, not via program variables. To rectify this, we can pretend that `t` and `v` *are* in the heap, at locations `&t` and `&v` respectively, and henceforth write assertions such as '$\mathtt{t} = 3$' instead as '$\mathtt{\&t} \mapsto 3$'. However, this practice requires a significant notational overhead; for instance, '$\mathtt{t} \geq 3$' becomes '$\exists T.\, \mathtt{\&t} \mapsto T * T \geq 3$'. Hence, having acknowledged that this is the proper way to handle mutable program variables, we shall revert to our previous practice, and implicitly understand that `t` and `v` actually refer to the heap.

4.3.8 The proof

Let C_m and C_f stand for the implementations of `malloc` and `free`. We are to prove:

$\mathtt{atomic}\, C_\mathsf{m} \quad \mathbf{sat}_{\mathsf{GSep}}$

PRE	GUAR	POST
$\left\lceil \begin{array}{l} \mathit{mod\_inv}\ \mathtt{s\,v\,t} \\ \vee\ \mathit{uninit}\ \mathtt{s} \end{array} \right\rceil_G$	G	$\left\lfloor \mathit{mod\_inv}\ \mathtt{s\,v\,t} \right\rfloor_G * \left(\begin{array}{l} \left(\left\lfloor \mathit{token}\ \mathtt{s\,v\,t\,ret}\ \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \right\rfloor_G \right. \\ \left. * \left\vert \begin{array}{l} \mathtt{ret} \\ \hline \\ \end{array} \right\vert^{\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil} \right) \\ \vee\ \mathtt{ret} \doteq 0 \end{array} \right)$

$\mathtt{atomic}\, C_\mathsf{f} \quad \mathbf{sat}_{\mathsf{GSep}}$

PRE	GUAR	POST
$\left\lceil \mathit{mod\_inv}\ \mathtt{s\,v\,t} \right\rceil_G * \left\lceil \mathit{token}\ \mathtt{s\,v\,t\,ap}\ n \right\rceil_G * \left\vert \begin{array}{l} \mathtt{ap} \\ \hline \\ \end{array} \right\vert^{\mathtt{ap}+n}$	G	$\left\lfloor \mathit{mod\_inv}\ \mathtt{s\,v\,t} \right\rfloor_G$

When verifying the module, we have the definitions of $\mathit{mod\_inv}$ and token available, so we can use the fact that they are both stable to erase the stabilisation brackets. We can also unfold the definitions of those predicates, such that, after some algebraic rearrangement, our goals become:

$\mathtt{atomic}\, C_\mathsf{m} \quad \mathbf{sat}_{\mathsf{GSep}}$

PRE	GUAR	POST
$\exists A.$ $\boxed{\begin{array}{l} \mathit{arena}\ \mathtt{s\,v\,t}\ A \\ \vee \left(\begin{array}{l} \mathit{uninit}\ \mathtt{s} \\ \wedge\ A = \emptyset \end{array} \right) \end{array}}$	G	$\exists A.$ $\boxed{\begin{array}{l} (\mathit{arena}\ \mathtt{s\,v\,t}\ A * \mathtt{ret} \doteq 0) \vee \\ \left(\mathit{arena}\ \mathtt{s\,v\,t}\ (A \uplus \{\mathtt{ret} \mapsto \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil\}) * \mathtt{ret} \dot{\neq} 0\right) \end{array}}$ $* \left(\left(\frac{1}{2} \left\vert \begin{array}{l} \mathtt{ret} - 1 \\ \hline (\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil)_\blacksquare \end{array} \right\vert * \left\vert \begin{array}{l} \mathtt{ret} \\ \hline \\ \end{array} \right\vert^{\mathtt{ret} + \lceil \mathtt{nb/WORD} \rceil} \right) \vee \mathtt{ret} \doteq 0 \right)$

$\mathtt{atomic}\, C_\mathsf{f} \quad \mathbf{sat}_{\mathsf{GSep}}$

PRE	GUAR	POST
$\exists A.$ $\boxed{\mathit{arena}\ \mathtt{s\,v\,t}\ (A \uplus \{\mathtt{ap} \mapsto n\})}$ $* \frac{1}{2} \left\vert \begin{array}{l} \mathtt{ap} - 1 \\ \hline (\mathtt{ap} + n)_\blacksquare \end{array} \right\vert * \left\vert \begin{array}{l} \mathtt{ap} \\ \hline \\ \end{array} \right\vert^{\mathtt{ap}+n}$	G	$\exists A.$ $\boxed{\mathit{arena}\ \mathtt{s\,v\,t}\ A}$

Next, we apply the GSEP-EXISTS rule. This eliminates all four occurrences of '$\exists A$' in the specifications. We can then apply the GSEP-ATOMIC rule. Note that each boxed assertion is

precise, as is required by the rule. We must confirm that the following state transitions are permitted by G.

$$\begin{pmatrix}(\mathit{uninit}\ \mathtt{s} \wedge A = \emptyset)\\ \vee\ \mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A\end{pmatrix} \rightsquigarrow \begin{pmatrix}(\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A * \mathtt{ret} \doteq 0) \vee\\ (\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \lceil \tfrac{\mathtt{nb}}{\mathtt{WORD}} \rceil\}) * \mathtt{ret} \dot{\neq} 0)\end{pmatrix} \quad (4.23)$$

$$\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ (A \uplus \{\mathtt{ap} \mapsto n\}) \rightsquigarrow \mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A \quad (4.24)$$

This is straightforward: (4.23) is an instance of either the $\mathit{Malloc}(\mathtt{ret})$ action or the Tau action, while (4.24) is an instance of Tau. Having unpacked the boxes, and performed some algebraic rearrangement, we are left to verify the following triples:

$$\left\{\begin{array}{l}(\mathit{uninit}\ \mathtt{s} \wedge A = \emptyset) \vee\\ \mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A\end{array}\right\}\ C_{\mathsf{m}}\ \left\{\begin{array}{l}(\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A * \mathtt{ret} \doteq 0) \vee\\ \left(\begin{array}{l}\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \lceil \tfrac{\mathtt{nb}}{\mathtt{WORD}} \rceil\}) * \mathtt{ret} \dot{\neq} 0\\ * \tfrac{1}{2}\left|\begin{array}{l}\mathtt{ret}-1\\ \hline (\mathtt{ret} + \lceil \mathtt{nb}/\mathtt{WORD} \rceil)_{\blacksquare}\end{array}\right| * \left|\begin{array}{l}\mathtt{ret}\\ \hline\end{array}\right|^{\mathtt{ret} + \lceil \mathtt{nb}/\mathtt{WORD} \rceil}\end{array}\right)\end{array}\right\}$$

$$\left\{\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ (A \uplus \{\mathtt{ap} \mapsto n\}) * \tfrac{1}{2}\left|\begin{array}{l}\mathtt{ap}-1\\ \hline (\mathtt{ap}+n)_{\blacksquare}\end{array}\right| * \left|\begin{array}{l}\mathtt{ap}\\ \hline\end{array}\right|^{\mathtt{ap}+n}\right\}\ C_{\mathsf{f}}\ \left\{\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A\right\}$$

To verify these triples, we revert now to proof outlines.

Proof outline for malloc

Note that the original source is available from the Unix Heritage Society [Bell Labs 1979].

```
1  #define WORD sizeof(union store)
2  #define BLOCK 1024 /* a multiple of WORD*/
3  #define testbusy(p) ((int)(p)&1)
4  #define setbusy(p) (st *)((int)(p)|1)
5  #define clearbusy(p) (st *)((int)(p)&~1)
6
7  struct store {struct store *ptr;};
8  typedef struct store st;
9  static st s[2]; /*initial arena*/
10 static st *v; /*search ptr*/
11 static st *t; /*arena top*/
12
13 char *malloc(unsigned int nb)
14 {
15   {(uninit s ∧ A = ∅) ∨ arena s v t A}
16   register st *p, *q;
17   register nw;
18   static temp;
```

19 `// Using Lem 4.10`

20 $\left\{\begin{array}{l}\exists n.\ \left|\frac{\mathtt{s}}{n}\right| * ((n \doteq 0 * \left|\frac{\mathtt{s}+1}{0}\right| * A \doteq \emptyset * brka(\mathtt{s}+2)) \vee \\ \qquad\qquad (n \dot{>} 0 * (\left|\frac{\mathtt{s}}{n}\right| \mathrel{-\!\!*} arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A)))\end{array}\right\}$

21 `if(s[0].ptr == 0) { /*first time*/`

22 $\left\{\left|\frac{s}{0}\middle|\frac{}{0}\right| * brka(\mathtt{s}+2) * A \doteq \emptyset\right\}$

23 `s[0].ptr = setbusy(&s[1]);`

24 $\left\{\left|\frac{s}{(s+1)_{\blacksquare}}\middle|\frac{}{0}\right| * brka(\mathtt{s}+2) * A \doteq \emptyset\right\}$

25 `s[1].ptr = setbusy(&s[0]);`

26 $\left\{\left|\frac{s}{(s+1)_{\blacksquare}}\middle|\frac{}{\mathtt{s}_{\blacksquare}}\right| * brka(\mathtt{s}+2) * A \doteq \emptyset\right\}$

27 `t = &s[1];`

28 $\left\{\left|\frac{\mathtt{s}}{\mathtt{t}_{\blacksquare}}\right| * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathtt{s} \dot{<} \mathtt{t} * brka(\mathtt{t}+1) * A \doteq \emptyset\right\}$

29 `v = &s[0];`

30 $\left\{\left|\frac{\mathtt{s}}{\mathtt{t}_{\blacksquare}}\right| * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathtt{s} \dot{<} \mathtt{t} * \mathtt{v} \doteq \mathtt{s} * brka(\mathtt{t}+1) * A \doteq \emptyset\right\}$

31 $\left\{chunk_{\mathtt{s}}\ \mathtt{v}\ \mathtt{t} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathtt{v} \doteq \mathtt{s} * brka(\mathtt{t}+1) * A \doteq \emptyset\right\}$

32 $\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * A \doteq \emptyset\end{array}\right\}$

33 $\{arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A * A \doteq \emptyset\}$

34 $\{arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A\}$

35 `} else {`

36 `// Using` $p * (p \mathrel{-\!\!*} q) \Rightarrow q$

37 $\{arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A\}$

38 `}`

39 $\left\{\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1)\right\}$

40 `nw=(nb+WORD+WORD-1)/WORD;`

41 $\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil\end{array}\right\}$

42 `for(p=v; ; ) {`

43 `// Loop inv 1:`

44 $\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * chunks\ \mathtt{p}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil\end{array}\right\}$

45 `for(temp=0; ; ) { // temp is needed only for total correctness`

46 `// Loop inv 2:`

47 $\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * chunks\ \mathtt{p}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil\end{array}\right\}$

48 `if(!testbusy(p->ptr)) {`

49 $\left\{\begin{array}{l}\exists C_1, C_2, r.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ r * \mathit{chunks}\ r\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

50 `  q = p->ptr;`

51 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * \mathit{chunks}\ \mathtt{q}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

52 `  while(!testbusy(q->ptr)) {`

53 $\left\{\begin{array}{l}\exists C_1, C_2, r.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * \mathit{chunk}_{\mathtt{u}}\ \mathtt{q}\ r * \mathit{chunks}\ r\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

54 `    p->ptr = q->ptr; // coalesce consecutive free chunks`

55 $\left\{\begin{array}{l}\exists C_1, C_2, r.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ r * \mathit{chunks}\ r\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

56 `    q = p->ptr;`

57 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * \mathit{chunks}\ \mathtt{q}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

58 `  }`

59 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * \mathit{chunks}\ \mathtt{q}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

60 `  if(q>=p+nw && p+nw>=p) { // integer overflows aren't modelled`

61 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunk}_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * \mathit{chunks}\ \mathtt{q}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{q} \geq \mathtt{p} + \mathtt{nw} \end{array}\right\}$

62 `    goto found;`

63 $\{\mathit{false}\}$

64 `  }`

65 `  // Using Lem 4.7`

66 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunks}\ \mathtt{p}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

67 `}`

68 `// p's chunk is unavailable / too small,`

69 `// or p points to the top of the arena`

70 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunks}\ \mathtt{p}\ \mathtt{t}\ C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil \end{array}\right\}$

71 `q = p;`

72 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{q}\ C_1 * \mathit{chunks}\ \mathtt{q}\ \mathtt{t}\ C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{q} \doteq \mathtt{p} \end{array}\right\}$

73 `p = clearbusy(p->ptr);`

74 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{q}\ C_1 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) \\ * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * ((\exists \tau.\ \mathit{chunk}_\tau\ \mathtt{q}\ \mathtt{p} * \mathit{chunks}\ \mathtt{p}\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \tau, \mathtt{p}\rangle])) \\ \vee\ (C_2 \doteq [] * \mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s}))\end{array}\right\}$

75 `if(p>q) {`

76 $\left\{\begin{array}{l}\exists C_1, C_2, \tau.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{q}\ C_1 * \mathit{chunk}_\tau\ \mathtt{q}\ \mathtt{p} * \mathit{chunks}\ \mathtt{p}\ \mathtt{t}\ (C_2 \circ\!\!- [\langle \mathtt{q}, \tau, \mathtt{p}\rangle]) \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil\end{array}\right\}$

77 `} else if(q!=t || p!=s) {`

78 $\left\{\begin{array}{l}\exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{q}\ C * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq C^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) \\ * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * \mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s} * (\mathtt{q} \dot{\neq} \mathtt{t} \vee \mathtt{p} \dot{\neq} \mathtt{s})\end{array}\right\}$

79 $\{\mathit{false}\}$

80 `return 0; // unreachable`

81 $\{\mathit{false}\}$

82 `} else if(++temp>1) {`

83 $\left\{\begin{array}{l}\exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{q}\ C * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq C^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) \\ * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * \mathtt{q} \doteq \mathtt{t} * \mathtt{p} \doteq \mathtt{s}\end{array}\right\}$

84 `break; // jump to [Extend arena]`

85 $\{\mathit{false}\}$

86 `}`

87 `// Reestablish loop inv 2:`

88 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{p}\ C_1 * \mathit{chunks}\ \mathtt{p}\ \mathtt{t}\ C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil\end{array}\right\}$

89 `}`

90 `// We never exit the loop 'normally' (because the non-existent`

91 `// test condition never fails). We only reach this point by`

92 `// breaking.`

93 `// [Extend arena]:`

94 $\left\{\exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{t}\ C * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq C^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * \mathtt{p} \doteq \mathtt{s}\right\}$

95 `temp = ((nw+BLOCK/WORD)/(BLOCK/WORD))*(BLOCK/WORD);`

96 $\left\{\begin{array}{l}\exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{t}\ C * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq C^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * \mathtt{p} \doteq \mathtt{s} \\ * \mathtt{temp} \dot{>} \mathtt{nw}\end{array}\right\}$

97 `q = (st *)sbrk(0);`

98 `// note that` $\mathit{brka}(\mathtt{q}) \Longrightarrow \mathit{brka}(\mathtt{t}+1)$ `by Lem 4.12`

99 $\left\{\begin{array}{l}\exists C.\ \mathit{chunks}\ \mathtt{s}\ \mathtt{t}\ C * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq C^{\mathtt{a}} * \mathit{brka}(\mathtt{t}+1) \\ * \mathtt{nw} \doteq 1 + \lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \rceil * \mathtt{p} \doteq \mathtt{s} * \mathtt{temp} \dot{>} \mathtt{nw} * \mathtt{q} \dot{\geq} \mathtt{t}+1\end{array}\right\}$

100 `if(q + temp < q) {`

101 $\{\mathit{false}\}$ `// integer overflows aren't modelled`

```
102       return 0;
103       {false}
104     }
105     { ∃C. chunks s t C * |t / s■| * A ≐ C^a * brka(t + 1)
          * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s * temp ≻ nw * q ≽ t + 1 }
106     q = (st *)sbrk(temp * WORD);
107     { ∃C. chunks s t C * |t / s■| * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s * temp ≻ nw
          * ((brka(t + 1) * q ≐ −1 / WORD) ∨ (brka(q + temp) * t + 1 ≤ q * |q ⟶ q + temp|)) }
108     if((int)q == -1) {
109       { ∃C. chunks s t C * |t / s■| * A ≐ C^a * brka(t + 1) }
110       v = s; // line added to fix bug
111       { ∃C1, C2. chunks s v C1 * chunks v t C2 * |t / s■| * A ≐ (C1 ∘ C2)^a * brka(t + 1) }
112       {arena A}
113       {(arena A * ret ≐ 0)[0/ret]}
114       return 0;
115       {false}
116     }
117     { ∃C. chunks s t C * |t / s■| * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s
          * temp ≻ nw * brka(q + temp) * t + 1 ≤ q * |q ⟶ q + temp| }
118     t->ptr = q;
119     { ∃C. chunks s t C * |t / q| * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s
          * temp ≻ nw * brka(q + temp) * t + 1 ≤ q * |q ⟶ q + temp| }
120     if(q!=t+1) {
121       { ∃C. chunks s t C * |t / q| * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s
          * temp ≻ nw * brka(q + temp) * t + 1 ≺ q * |q ⟶ q + temp| }
122       t->ptr = setbusy(t->ptr);
123       { ∃C. chunks s t C * |t / q■| * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉ * p ≐ s
          * temp ≻ nw * brka(q + temp) * t + 1 ≺ q * |q ⟶ q + temp| }
124       { ∃C. chunks s t C * chunk_s t q * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉
          * p ≐ s * temp ≻ nw * brka(q + temp) * |q ⟶ q + temp| }
125     }
126     // t is either an unallocated chunk (of size 0)
127     // or a system-allocated chunk
128     { ∃C. chunks s t C * (chunk_u t q ∨ chunk_s t q) * A ≐ C^a * nw ≐ 1 + ⌈nb / WORD⌉
          * p ≐ s * temp ≻ nw * brka(q + temp) * |q ⟶ q + temp| }
```

```
129     // C swallows the chunk at t. A = C^a still holds because
130     // the chunk at t isn't allocated to a client.
```

$$\left\{\begin{array}{l}\exists C.\ chunks\ \mathtt{s}\ \mathtt{q}\ C * A = C^{\mathtt{a}} * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{p} \doteq \mathtt{s} * \mathtt{temp} \dot{>} \mathtt{nw} \\ * \ brka(\mathtt{q} + \mathtt{temp}) * \left|\frac{\mathtt{q}}{\ }\right| * \ {}^{\mathtt{q}}\left|\text{——}\right|^{\mathtt{q}+\mathtt{temp}-1} * \left|\frac{\mathtt{q}+\mathtt{temp}-1}{\ }\right|\end{array}\right\}$$

```
132     t = q->ptr = q+temp-1;
```

$$\left\{\begin{array}{l}\exists C.\ chunks\ \mathtt{s}\ \mathtt{q}\ C * A \doteq C^{\mathtt{a}} * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{p} \doteq \mathtt{s} \\ * \ brka(\mathtt{t}+1) * \left|\frac{\mathtt{q}}{\mathtt{t}}\right| * \ {}^{\mathtt{q}}\left|\text{——}\right|^{\mathtt{t}} * \left|\frac{\mathtt{t}}{\ }\right|\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists C.\ chunks\ \mathtt{s}\ \mathtt{q}\ C * A \doteq C^{\mathtt{a}} * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{p} \doteq \mathtt{s} \\ * \ brka(\mathtt{t}+1) * chunk_{\mathtt{u}}\ \mathtt{q}\ \mathtt{t} * \left|\frac{\mathtt{t}}{\ }\right|\end{array}\right\}$$

```
135     // C swallows the chunk at q. A = C^a still holds because
136     // the chunk at q isn't allocated.
```

$$\left\{\exists C.\ chunks\ \mathtt{s}\ \mathtt{t}\ C * A \doteq C^{\mathtt{a}} * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{p} \doteq \mathtt{s} * brka(\mathtt{t}+1) * \left|\frac{\mathtt{t}}{\ }\right|\right\}$$

```
138     t->ptr = setbusy(s);
139     // reestablish loop inv 1:
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * chunks\ \mathtt{p}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * \ A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil\end{array}\right\}$$

```
141 }
```

$$\{false\}$$

```
143 found:
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * chunk_{\mathtt{u}}\ \mathtt{p}\ \mathtt{q} * chunks\ \mathtt{q}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * \ A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{q} \dot{\geq} \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
145     v = p+nw;
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{q}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}}\left|\text{——}\right|^{\mathtt{q}} * chunks\ \mathtt{q}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * \ A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
147     if (q>v) {
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{q}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}}\left|\text{——}\right|^{\mathtt{q}} * chunks\ \mathtt{q}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * \ A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
149       v->ptr = p->ptr;
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{q}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}} * chunk_{\mathtt{u}}\ \mathtt{v}\ \mathtt{q} * chunks\ \mathtt{q}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * \ A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{q}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}} * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
152     }
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{q}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}} * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil \frac{\mathtt{nb}}{\mathtt{WORD}} \right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
154   p->ptr = setbusy(v);
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * \left|\frac{\mathtt{p}}{\mathtt{v}_{\blacksquare}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}} * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil\frac{\mathtt{nb}}{\mathtt{WORD}}\right\rceil * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{p}\ C_1 * chunk_{\mathtt{a}}\ \mathtt{p}\ \mathtt{v} * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \doteq (C_1 \circ C_2)^{\mathtt{a}} \\ * brka(\mathtt{t}+1) * \mathtt{nw} \doteq 1 + \left\lceil\frac{\mathtt{nb}}{\mathtt{WORD}}\right\rceil * \frac{1}{2}\left|\frac{\mathtt{p}}{\mathtt{v}_{\blacksquare}}\right| * \ {}^{\mathtt{p}}\left|\text{——}\right|^{\mathtt{v}} * \mathtt{v} \doteq \mathtt{p} + \mathtt{nw}\end{array}\right\}$$

```
157   // Use Lem 4.7 to combine C1 with the chunk at p.
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * A \uplus \{\mathtt{p}+1 \mapsto \left\lceil\frac{\mathtt{nb}}{\mathtt{WORD}}\right\rceil\} \doteq (C_1 \circ C_2)^{\mathtt{a}} \\ * brka(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{p}}{(\mathtt{p}+\lceil \mathtt{nb}/\mathtt{WORD}\rceil+1)_{\blacksquare}}\right| * \left|\frac{\mathtt{p}+1}{}\right|^{\mathtt{p}+1+\lceil \mathtt{nb}/\mathtt{WORD}\rceil}\end{array}\right\}$$

$$\left\{\left(\begin{array}{l}arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \left\lceil\frac{\mathtt{nb}}{\mathtt{WORD}}\right\rceil\}) \\ * \left|\frac{\mathtt{ret}}{}\right|^{\mathtt{ret}+\lceil \mathtt{nb}/\mathtt{WORD}\rceil} * \frac{1}{2}\left|\frac{\mathtt{ret}-1}{(\mathtt{ret}+\lceil \mathtt{nb}/\mathtt{WORD}\rceil)_{\blacksquare}}\right|\end{array}\right)[\mathtt{p}+1/\mathtt{ret}]\right\}$$

```
160   return((char *)(p+1));
```

$$\{false\}$$

```
162 }
```

$$\left\{\left(\begin{array}{l}arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ret} \mapsto \left\lceil\frac{\mathtt{nb}}{\mathtt{WORD}}\right\rceil\}) \\ * \left|\frac{\mathtt{ret}}{}\right|^{\mathtt{ret}+\lceil \mathtt{nb}/\mathtt{WORD}\rceil} \\ * \frac{1}{2}\left|\frac{\mathtt{ret}-1}{(\mathtt{ret}+\lceil \mathtt{nb}/\mathtt{WORD}\rceil)_{\blacksquare}}\right|\end{array}\right) \vee (arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ A * \mathtt{ret} \doteq 0)\right\}$$

Proof outline for free

```
164 free(register char *ap)
165 {
```

$$\left\{arena\ \mathtt{s}\ \mathtt{v}\ \mathtt{t}\ (A \uplus \{\mathtt{ap} \mapsto n\}) * \frac{1}{2}\left|\frac{\mathtt{ap}-1}{(\mathtt{ap}+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{ap}}{}\right|^{\mathtt{ap}+n}\right\}$$

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ \mathtt{v}\ C_1 * chunks\ \mathtt{v}\ \mathtt{t}\ C_2 * A \uplus \{\mathtt{ap} \mapsto n\} \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * brka(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{ap}-1}{(\mathtt{ap}+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{ap}}{}\right|^{\mathtt{ap}+n}\end{array}\right\}$$

```
168   // Use Lem 4.7 to combine C1 and C2
```

$$\left\{\begin{array}{l}\exists C.\ chunks\ \mathtt{s}\ \mathtt{t}\ C * A \uplus \{\mathtt{ap} \mapsto n\} \doteq C^{\mathtt{a}} \\ * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * brka(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{ap}-1}{(\mathtt{ap}+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{ap}}{}\right|^{\mathtt{ap}+n}\end{array}\right\}$$

```
170   // Since ⟨ap, a, n⟩ ∈ C we can use Lem 4.11 to split C
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ (\mathtt{ap}-1)\ C_1 * chunk_{\mathtt{a}}\ (\mathtt{ap}-1)\ (\mathtt{ap}+n) \\ * chunks\ (\mathtt{ap}+n)\ \mathtt{t}\ C_2 * A \uplus \{\mathtt{ap} \mapsto n\} \doteq (C_1 \circ \langle \mathtt{ap}, \mathtt{a}, n\rangle \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| \\ * brka(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{ap}-1}{(\mathtt{ap}+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{ap}}{}\right|^{\mathtt{ap}+n}\end{array}\right\}$$

```
172   // cancel ap chunk from both sides
```

$$\left\{\begin{array}{l}\exists C_1, C_2.\ chunks\ \mathtt{s}\ (\mathtt{ap}-1)\ C_1 * chunk_{\mathtt{a}}\ (\mathtt{ap}-1)\ (\mathtt{ap}+n) * chunks\ (\mathtt{ap}+n)\ \mathtt{t}\ C_2 \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * brka(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{ap}-1}{(\mathtt{ap}+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{ap}}{}\right|^{\mathtt{ap}+n}\end{array}\right\}$$

```
174   register st *p = (st *)ap;
```

175 `v = --p;`

176 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\,\mathtt{p}\,C_1 * \mathit{chunk}_{\mathtt{a}}\ \mathtt{p}\,(\mathtt{p}+1+n) * \mathit{chunks}\,(\mathtt{p}+1+n)\,\mathtt{t}\,C_2 \\ * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \frac{1}{2}\left|\frac{\mathtt{p}}{(\mathtt{p}+1+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{p}+1}{}\right|^{\mathtt{p}+n+1} * \mathtt{p} \doteq \mathtt{v}\end{array}\right\}$

177 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\,\mathtt{p}\,C_1 * \left|\frac{\mathtt{p}}{(\mathtt{p}+1+n)_{\blacksquare}}\right| * \left|\frac{\mathtt{p}+1}{}\right|^{\mathtt{p}+n+1} \\ * \mathit{chunks}\,(\mathtt{p}+1+n)\,\mathtt{t}\,C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{p} \doteq \mathtt{v}\end{array}\right\}$

178 `p->ptr = clearbusy(p->ptr);`

179 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\,\mathtt{p}\,C_1 * \left|\frac{\mathtt{p}}{\mathtt{p}+1+n}\right| * \left|\frac{\mathtt{p}+1}{}\right|^{\mathtt{p}+n+1} \\ * \mathit{chunks}\,(\mathtt{p}+1+n)\,\mathtt{t}\,C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{p} \doteq \mathtt{v}\end{array}\right\}$

180 $\left\{\begin{array}{l}\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\,\mathtt{p}\,C_1 * \mathit{chunk}_{\mathtt{u}}\,\mathtt{p}\,(\mathtt{p}+1+n) \\ * \mathit{chunks}\,(\mathtt{p}+1+n)\,\mathtt{t}\,C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1) * \mathtt{p} \doteq \mathtt{v}\end{array}\right\}$

181 *// Use Lem 4.7 to combine the chunk at p with* C_2.

182 $\left\{\exists C_1, C_2.\ \mathit{chunks}\ \mathtt{s}\,\mathtt{v}\,C_1 * \mathit{chunks}\ \mathtt{v}\,\mathtt{t}\,C_2 * A \doteq (C_1 \circ C_2)^{\mathtt{a}} * \left|\frac{\mathtt{t}}{\mathtt{s}_{\blacksquare}}\right| * \mathit{brka}(\mathtt{t}+1)\right\}$

183 $\{\mathit{arena}\ \mathtt{s}\,\mathtt{v}\,\mathtt{t}\ A\}$

184 `}`

4.4 Remarks about the proof

An early version of the proof outline presented above was produced by Mike Dodds, who discovered the bug described in Sect. 4.3.4. An amended version of his proof can be found in [Wickerson et al. 2010b]. Since his initial version, the proof has been adapted in several important ways. Dodds' proof also used insights from RGSep, but did not treat the procedure bodies as atomic. His RGSep actions are therefore much more fine-grained. As a result of this, his proof is unable to handle the initialisation phase (lines 19–38), nor the bug-fix added on line 110. In moving to an atomic treatment of procedure bodies, it is necessary to parameterise the *chunks* predicate by the list C of chunks that it contains.

The proof presented above is large and complex. Because it has not been mechanically checked, it is likely to contain some mistakes, but we expect that these will be minor. The level of detail shown is quite variable. Between lines 179 and 180, for instance, very little has changed; we have merely folded the definition of $\mathit{chunk}_{\mathtt{u}}$. Line 50, on the other hand, is comparatively involved: in order to dereference `p` we are required to unfold (and later re-fold) the $\mathit{chunk}_{\mathtt{u}}$ predicate so that the cell at `p` is exposed. Some particularly tricky steps have been adorned with textual explanations. This practice is flawed in general, because such text can become desynchronised with a developing proof, just as comments can in source code. How, then, can the presentation of such proofs be improved?

The use of a theorem prover would certainly address any concerns about errors in the proof. But even if the proof above *had* been mechanically checked, it would remain hard to read and understand. This presentation is little more than a long list of assertions with some interspersed lines of code. The problem with this is that the only way to add more detail is to add more assertions. More assertions bring more repetition, and the more repetitive the proof is, the harder it is to read. Perhaps even more importantly, such a repetitive proof is hard to change, since even minor edits must be propagated through a great number of similar assertions.

In the next chapter we shall delve further into the issues surrounding representations of program proofs, and propose a better way to depict the proof above.

4.5 Related and future work

In this chapter, we have considered the applicability of RGSep to the verification of sequential modules, and the role of explicit stabilisation in improving the modularity of this verification.

GSep continues a tradition of mutual inheritance between sequential and concurrent program logics. The *assume-guarantee* technique, for instance, was devised by Misra and Chandy [1981] for message-passing concurrency, but more recently applied to sequential heap-manipulating procedures by Yorsh et al. [2005]. In the realm of separation logic, the hypothetical frame rule [O'Hearn et al. 2004] can be understood as a sequential version of concurrent separation logic [O'Hearn 2004; Brookes 2004]; the common factor being the notion of a resource invariant. In the hypothetical frame rule, the resource invariant represents the state of the module (which is only available from within the module's procedures) while in concurrent separation logic it is recast as the shared state (which is only available from within an atomic block). RGSep can be viewed as a refinement of concurrent separation logic, and GSep can be viewed as a refinement of the hypothetical frame rule in the same way.

	for concurrency	for sequential modules
uses invariants	concurrent separation logic	hypothetical frame rule
uses relations	RGSep	GSep

(4.25)

GSep is more expressive than the hypothetical frame rule for the same reason that RGSep is more expressive than concurrent separation logic (unless auxiliary state is employed). In concurrent separation logic, from a program point outside of an atomic block, the only assertion that can be made about the shared state is that the resource invariant holds. In RGSep, however, we may make any assertion at any point about the shared state, providing that the assertion is stable.

It is for this reason that GSep proves necessary for verifying the Version 7 Unix memory manager, and why the hypothetical frame rule alone is insufficient. After a call to `malloc`, it is not enough simply to assert that the module state continues to satisfy an invariant: we must additionally know that the module state contains a suitably-sized and suitably-positioned 'gap', into which the `malloc`'d chunk can later be `free`'d. This pattern, of requiring limited knowledge of the module's internal configuration, is surely common to many modules, and we therefore argue that GSep occupies an important point in the program logic design space.

That said, there are several shortcomings in our approach, three of which we describe below, together with possible fixes.

Beyond simple modules?

Our approach handles only simple modules. That is, we work with a programming language in which modules may only be declared at the top level of a program, while other authors, such as O'Hearn et al. [2004], consider more general languages that support locally-declared modules. In fact, our programs can declare only a *single* module, but this is not an important restriction, as we can extend GSep to handle multiple top-level modules in the same way that Vafeiadis [2007, §4.3.2] extends RGSep to handle multiple shared regions. The restriction to top-level modules, however, is non-trivial. It is inherited from an inherent limitation of RGSep, pointed out by Feng [2009]: that it 'require[s] the shared resource be globally known.' In RGSep, each part of the heap is either entirely private to one thread or entirely shared among every thread. A thread cannot, for instance, divide into two child threads and give to them part of its local heap to share. As a result, RGSep cannot verify programs whose shared regions are locally-scoped, and GSep cannot verify programs whose modules are locally-declared.

Feng [2009] and Dinsdale-Young et al. [2010] both propose refinements to RGSep that fix this problem. Where RGSep uses boxes to delimit those parts of an assertion that refer to shared regions, Feng's *local rely-guarantee* (LRG) couples each assertion with a 'shared-state invariant'. This is an assertion responsible for picking out the shared region. The use of invariants enables the definition of a $*$-operator on relies and guarantees, through which interference can be limited to a small number of threads. Dinsdale-Young et al.'s system of *concurrent abstract predicates* (CAP) also defines a $*$-operator on relies and guarantees, but retains the boxed assertions from RGSep. It introduces a 're-partitioning implication' that allows the local/shared division of the state to be modified dynamically. Through this operator, new shared regions can be created for just one part of a program.

Both LRG and CAP could be recast as logics for sequential modules, thus adding further rows to the table in (4.25). LRG, in its current form, is not suitable for verifying the Version 7 Unix memory manager, as its assertion language does not permit the same piece of state to be described multiple times. RGSep and CAP both allow this, and hence enable several clients each to make an assertion about the module state. A sequential version of CAP, on the other hand, would be well-suited to verifying our memory manager as part of a hierarchy of locally-declared modules. Moreover, CAP can be easily extended from 'statically-scoped' (parallel composition) concurrency to 'dynamically-scoped' (fork and join) concurrency, as shown by Dodds et al. [2009]. Therefore, a sequential version of CAP would be appropriate when considering extensions of this work from static modules (such as the memory manager) to dynamic modules (i.e. objects).

Nonetheless, CAP is a rather complex logic. A rather more abstract alternative called *fictional separation logic* has been recently proposed by Jensen and Birkedal [2012]. Fictional separation logic takes a more abstract view of separation than the 'physical' separation of heap cells advocated by early work on separation logic. In the context of the Version 7 Unix memory manager, the assertions of two different clients do not describe physically separate regions of memory, since both refer to the module state. Nonetheless, since these clients are accessing different chunks in the arena, there is a sense in which these assertions are 'logically' separate, and hence can safely be treated as independent. To verify a module in fictional separation logic, one must find a sense in which its clients' assertions are separate, and phrase this as a commutative monoid. Krishnaswami et al. [2012] explain how to do just this for a simplified version of our memory manager, inspired by the proof we have presented in this chapter. The essence of their proof is broadly similar to our RGSep-based proof, but theirs has the key advantage of extending to multiple locally-scoped modules. On the other hand, it is possible that tool support would be easier to design for our RGSep-based approach, as rely-guarantee relations may be easier to infer automatically (e.g. using the technique of Vafeiadis [2010]) than commutative monoids.

Can we remove the module invariant?

A shortcoming of our approach is the presence of the mod_inv predicate throughout the specifications. We suspect that the hypothetical frame rule can be used to remove this predicate, but work remains to evaluate its soundness. The hypothetical frame rule is known not to be sound in concurrent separation logic in general (in case two threads both seek to hide the same invariant) but since we are not considering parallel composition in our context, it may well be in sound in GSep.

Can we remove the explicit stabilisation?

Another shortcoming in our approach is the presence of the explicit stabilisation operators in the client-facing specifications. Although explicit stabilisation has been useful for hiding the value of G from clients, it is nonetheless unsatisfying that stability has to be considered at all while verifying clients. We would prefer the client verification in Fig. 4.4 to look exactly like that in Fig. 4.1, and not have to feature any stabilisation brackets. We are almost there, but not quite. The GSep specifications we give for `malloc` and `free` do not feature any boxed assertions, so one might expect to be able to treat them as sequential. This is not so, because they contain abstract predicates, which may denote boxed assertions. In future work we intend to investigate the conditions under which abstract predicates ranging over GSep assertions can be safely replaced with those that range over ordinary separation logic assertions, and hence for our GSep specifications to become the ordinary separation logic specifications originally proposed by Parkinson and Bierman [2005].

Although a future refinement of this work may succeed in removing the explicit stabilisation from these specifications, we believe that explicit stabilisation remains an important technique in verification, because it is a useful and rigorous way to reason about the central issue of stability. Many program logics based on rely-guarantee have recently been devised – LRG, Deny-Guarantee [Dodds et al. 2009], and the logic of Gotsman et al. [2009] for proving liveness, to name a few – and the notions of stability in such logics are becoming ever more subtle. Stability is a central notion even in fields outside of software verification, such as biological systems [Cook et al. 2011]. It is therefore increasingly important to have a solid basis upon which to reason about stability, and we believe explicit stabilisation provides such a basis.

4.5.1 Alternative specifications for `malloc` and `free`

A significant source of complexity in both the verification and the use of our specifications for `malloc` and `free` is that we use RGSep to find implementations for the mod_inv and $token$ predicates. The use of RGSep then necessitates the introduction of a guarantee relation, which in turn makes the explicit stabilisation operators necessary in order to attain modularity.

We can avoid the use of RGSep altogether by adopting specifications for `malloc` and `free` that are different from those proposed by Parkinson and Bierman (see (4.1)), in which the set of allocated chunks is exposed to the client.

$$\left\{\begin{array}{l}(uninit \wedge A = \emptyset)\\ \vee\ arena(A)\end{array}\right\}\ \texttt{malloc(nb)}\ \left\{arena(A \uplus \{\texttt{ret} \mapsto \lceil \tfrac{\texttt{nb}}{\texttt{WORD}} \rceil\}) * \left|\begin{array}{|c|c|}\hline \texttt{ret} & \\ \hline & \\ \hline\end{array}\right|^{\texttt{ret} + \lceil \texttt{nb/WORD} \rceil}\right\}$$
$$\left\{arena(A \uplus \{\texttt{ap} \mapsto n\}) * \left|\begin{array}{|c|c|}\hline \texttt{ap} & \\ \hline & \\ \hline\end{array}\right|^{\texttt{ap}+n}\right\}\ \texttt{free(ap)}\ \left\{arena(A)\right\} \tag{4.26}$$

The effect of these specifications is to delegate to the client the task of monitoring the set of allocated chunks. We eliminate the distinction between the mod_inv and the $token$ predicates, and instead have just an $arena$ predicate, which is parameterised by a mapping A that associates the address of each chunk with its size.

These specifications are fairly acceptable alternatives, and are certainly simpler to verify against, as there is no need to introduce RGSep. Moreover, it turns out that `malloc` no longer needs to hand to each client half of the pointer preceding the newly-allocated chunk. As a result, we no longer need a separate class of 'chunks allocated to the system', which were previously expressed using the $chunk_{\mathtt{s}}$ predicate. For such chunks the $chunk_{\mathtt{a}}$ predicate is now sufficient.

Nevertheless, these specifications have several subtle flaws that make them less appealing from the client's perspective. We describe these flaws below, and illustrate them with the aid of

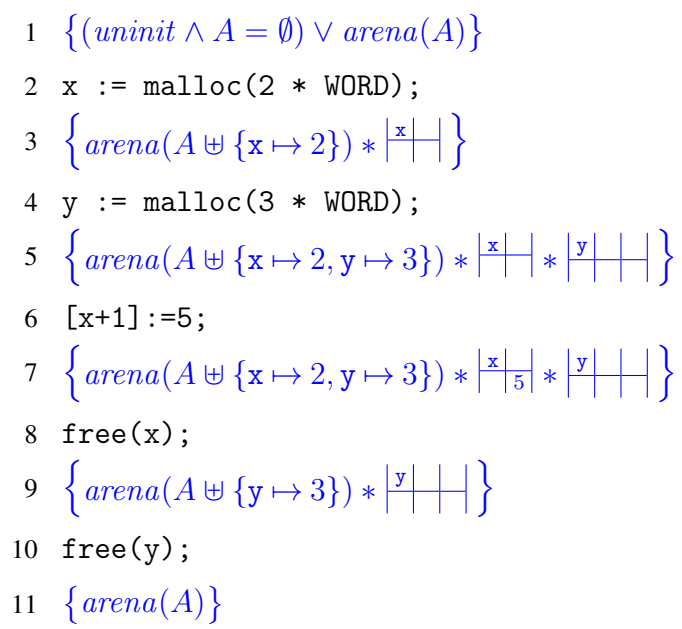


Figure 4.5: Proof outline of a simple client using the specifications in (4.26)

a proof outline, shown in Fig. 4.5, for a simple client that uses these specifications.

First, we have no hope of using the hypothetical frame rule to hide the arena, because there is no $*$-conjunct that is common to both the precondition and the postcondition. The memory manager is a static module, so the arena is present throughout the lifetime of the client, but by writing the specifications in this form, we cannot erase the arena. With the token-based approach, we may be able to use the hypothetical frame rule to hide the mod_inv predicate.

Second, it is rather unsatisfactory to have to thread the logical variable A throughout all client proofs. The token-based approach does not expose A to clients.

Third, these specifications must be used in a linear fashion. The $arena$ predicate must be in the precondition of every module call, and since it cannot be divided, these calls cannot occur in parallel. Granted, this module is not designed to be used in a concurrent context, but the fact that the $arena$ predicate must be threaded through all calls to `malloc` or `free` suggests more dependencies between these calls than those that actually exist. Conversely, as discussed above, the token-based approach potentially allows the mod_inv predicate to be hidden, and hence for these extraneous dependencies to be removed.

Fourth, in the token-based approach, one can distribute tokens throughout a data structure as it is created. This makes the later destruction of that data structure straightforward. To see this, consider a binary tree, whose nodes comprise three cells: a value v, and two pointers l and r to sub-trees. If we define $tree$ as the smallest predicate satisfying

$$tree(x) \quad \Leftrightarrow \quad (x = 0 \wedge emp) \vee (\exists v, l, r.\, x \mapsto v\, l\, r * token\, x\, 3 * tree(l) * tree(r))$$

then a recursive routine for tree disposal can be verified naturally – see Fig. 4.6. In contrast, the specifications in (4.26) would make verifying this routine much more cumbersome.

4.6 Conclusion

In this chapter and the last, we have proposed explicit stabilisation as a new way to deal with stability in rely-guarantee reasoning. The central idea is to record information about an assertion's

```
1  dispose_tree(x) {
2    {tree(x) * mod_inv}
3    if (x==0) {
4      {mod_inv}
5    } else {
6      {∃l, r. x ↦ _ l r * token x 3 * tree(l) * tree(r) * mod_inv}
7      y := [x+1];
8      z := [x+2];
9      {x ↦ _ y z * token x 3 * tree(y) * tree(z) * mod_inv}
10     dispose_tree(y);
11     {x ↦ _ y z * token x 3 * tree(z) * mod_inv}
12     dispose_tree(z);
13     {x ↦ _ y z * token x 3 * mod_inv}
14     free(x);
15     {mod_inv}
16   }
17   {mod_inv}
18 }
```

Figure 4.6: Proof outline of tree disposal

stability into its syntactic form. The main benefits are in modular reasoning:

Library code can be verified independently of clients. In Sect. 3.3, we showed how an approach based upon explicit stabilisation enables rely-guarantee reasoning to verify concurrent library code. Essentially, the stabilisation in the library's specification is evaluated so lazily that it actually becomes an obligation of the client.

Client code can be verified independently of a sequential module. We showed in this chapter how the application of explicit stabilisation to RGSep gives rise to a proof rule that allows a sequential module to hide its internal interference from its clients. Such information hiding is crucial for modular reasoning, because it allows the specification of a client to be reused, even despite changes to the specification of this internal interference. We demonstrated this reasoning by verifying a memory manager.

It would be interesting to investigate whether these two forms of modularity can be combined; that is, can we verify both a library and its clients, modularly, at the same time? It looks feasible. The specification for the library in Sect. 3.3 used explicit stabilisation with an arbitrary rely R, which became specific for each client in turn. Meanwhile, the specifications for the memory manager in Sect. 4.3.6 used explicit stabilisation with the specific G of the module, which was then generalised to an arbitrary G for the clients, so as to provide information hiding. Perhaps a combination of these approaches would parameterise on both the rely and the guarantee?

Chapter 5

Ribbon proofs for separation logic

We present ribbon proofs*, a diagrammatic proof system for separation logic. Inspired by an eponymous system due to Bean, ribbon proofs emphasise the structure of a proof, so are intelligible and hence useful pedagogically. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they are highly scalable (and we illustrate this with a ribbon proof of the Version 7 Unix memory manager). Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs. This chapter introduces the ribbon proof system, proves its soundness and completeness, and outlines a prototype tool for validating the diagrams in Isabelle.*

This chapter is based on a paper co-authored by Mike Dodds and Matthew Parkinson, presented in short form at LICS 2012, and published in the proceedings of ESOP 2013 [Wickerson et al. 2013].

5.1 Introduction

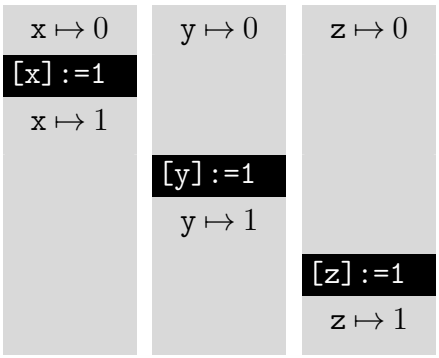
A program proof should not merely certify *that* a program is correct; it should explain *why* it is correct. A proof should be more than 'true': it should be informative, and it should be intelligible. In this chapter, we move away from investigating new methods for proving more properties of more programs, but rather, propose a new way to present such proofs. Building on work by Bean [2005], we describe a system that produces program proofs that are readable, scalable, and easily modified.

A program proof in Hoare logic is usually presented as a *proof outline*, in which the program's instructions are interspersed with 'enough' assertions to allow the reader to reconstruct the derivation tree. Since emerging circa 1971, the proof outline has become the de facto standard in the literature on both Hoare logic (e.g. [Hoare 1971a; Ashcroft 1976; Owicki and Gries 1976; Misra and Chandy 1981; Schneider 1997]) and its recent descendant, separation logic (e.g. [Ishtiaq and O'Hearn 2001; Reynolds 2002; Berdine et al. 2005; Bornat et al. 2005; Feng et al. 2007; Gotsman et al. 2007; Vafeiadis and Parkinson 2007; Dodds et al. 2009; Feng 2009; Dinsdale-Young et al. 2010; Hur et al. 2011; Jacobs et al. 2011b; Bornat and Dodds 2012]). Its great triumph is what might be called *instruction locality*: that one can verify each instruction in isolation (by confirming that the assertions immediately above and below it form a valid Hoare triple) and immediately deduce that the entire proof is correct.

Yet proof outlines suffer several shortcomings, some of which are manifested in Fig. 5.1a. This proof outline, first discussed in Sect. 2.5, concerns a program that writes to three memory

1 $\{x \mapsto 0 * y \mapsto 0 * z \mapsto 0\}$
2 [x]:=1;
3 $\{x \mapsto 1 * y \mapsto 0 * z \mapsto 0\}$
4 [y]:=1;
5 $\{x \mapsto 1 * y \mapsto 1 * z \mapsto 0\}$
6 [z]:=1;
7 $\{x \mapsto 1 * y \mapsto 1 * z \mapsto 1\}$

(a) A proof outline



(b) A ribbon proof

Figure 5.1: A simple example

cells; separation logic's $*$-operator specifies that these cells are distinct. First, there is much repetition: '$x \mapsto 1$' appears three times. Second, it is difficult to interpret the effect of each instruction because there is no distinction between those parts of an assertion that are actively involved and those that are merely in what separation logic calls the *frame*. For instance, line 4 affects only the second conjunct of its preceding assertion, but it is difficult to deduce the assignment's effect because two unchanged conjuncts are also present. These are only minor problems in our toy example, but they quickly become devastating when scaled to larger programs.

The crux of the problem is what might be called *resource locality*. Separation logic specialises in this second dimension of locality. As explained in Sect. 2.5, one can use separation logic's *small axioms* to reason about each instruction as if it were executing in a state containing only the resources (i.e. memory cells) that it needs, and immediately deduce its effect on the entire state using the FRAME rule. The proof outline below depicts this mechanism for line 4 of Fig. 5.1a.

$$\text{FRAME}\ \frac{x \mapsto 1 * z \mapsto 0 \quad \dfrac{\begin{array}{l}\{x \mapsto 1 * y \mapsto 0 * z \mapsto 0\}\\ \{y \mapsto 0\}\\ \quad \texttt{[y]:=1;}\\ \{y \mapsto 1\}\\ \{x \mapsto 1 * y \mapsto 1 * z \mapsto 0\}\end{array}}{\text{HEAPUPDATE}}}{}$$

Showing such detail throughout a proof outline would clarify the effect of each instruction, but escalate the repetition. Cleverer use of the FRAME rule can help, but only a little – see §5.7. Essentially, we need a new proof representation to harness the new technology separation logic provides, and we propose the **ribbon proof**.

Figure 5.1b gives an example. The repetition has disappeared, and each instruction's effect is now clear: it affects exactly those assertions directly above and below it, while framed assertions (which must not mention variables written by the instruction) pass unobtrusively to the left or right. Technically, we still invoke the FRAME rule at each instruction, but crucially in a ribbon proof, such invocations are implicit and do not complicate the diagram.

A bonus of this particular ribbon proof is that it emphasises that the three assignments update different memory cells. They are thus independent, and amenable to reordering or parallelisation. One can imagine obtaining a proof of the transformed program by simply sliding the left-hand column downward and the right-hand column upward. The corresponding proof outline neither suggests nor supports such manoeuvres.

Where a proof outline essentially flattens a proof to a list of assertions and instructions, our

system produces geometric objects that can be navigated and modified by leveraging human visual intuition, and whose basic steps correspond exactly to separation logic's small axioms. A ribbon proof de-emphasises the program's shallow syntax, such as the order of independent instructions, and illuminates instead the deeper structure, such as the flow of resources through the code. Proof outlines focus on Hoare triples $\{p\}\, c\, \{q\}$, and often neglect the details of entailments between assertions, $p \Rightarrow q$, even though such entailments often encode important insights about the program being verified. Ribbon proofs treat both types of judgement equally, within the same system.

There are many recent extensions of separation logic (e.g. [Feng et al. 2007; Gotsman et al. 2007; O'Hearn 2007; Vafeiadis and Parkinson 2007; Dodds et al. 2009; Feng 2009; Dinsdale-Young et al. 2010; Hur et al. 2011; Jacobs et al. 2011b]) to which our ribbon proof technology can usefully be applied; indeed, ribbons have already aided the development of a separation logic for relaxed memory [Bornat and Dodds 2012]. All of these program logics are based on increasingly complex reasoning principles, of which clear explanations are increasingly vital. We propose ribbon proofs as the ideal device for providing them.

The contributions made in this chapter are as follows. We describe a diagrammatic proof system that enables a natural presentation of separation logic proofs. Section 5.3 formally defines a two-dimensional language of ribbon diagrams and provides proof rules that are sound and complete with respect to separation logic. Section 5.4 gives an alternative, graphical formalisation that is sound if we remove the side-condition on the FRAME rule (by, for instance, using the *variables-as-resource* paradigm).

Because ribbon proofs contain much less redundancy than proof outlines, they are a more scalable proof representation. To illustrate the ability of our diagrams to present readable proofs of more complex programs, Sect. 5.5 presents a ribbon proof of the memory manager from Version 7 Unix, which was previously studied in Chapter 4.

We describe, in Sect. 5.6, a prototype tool for mechanically checking ribbon proofs in Isabelle (including several presented in this paper). Given a small proof script for each basic step, our tool assembles a script that verifies the entire diagram. Such tediums as the associativity and commutativity of the $*$-operator are handled in the graphical structure, leaving the user to focus on the interesting parts of the proof.

Comparison with Bean's system Bean [2005] introduced ribbon proofs as an extension of Fitch's *box proofs* [Fitch 1952] to handle the propositional fragment of bunched implications logic (BI) [O'Hearn and Pym 1999]. BI being the basis of the assertion language used in separation logic [Ishtiaq and O'Hearn 2001], his system can be used to prove entailments between propositional separation logic assertions. Our system expands Bean's into a full-blown program logic by adding support for commands and existentially-quantified variables. It is further distinguished by its treatment of ribbon proofs as graphs, which gives our diagrams an appealing degree of flexibility.

5.2 Anatomy of a ribbon proof

We describe our ribbon proof system using two examples.

```
1  {ls x 0 * ls y 0}
2  if (x==0) {
3    {ls y 0}
4    x:=y;
5    {ls x 0}
6  } else {
7    {ls x x * ls x 0 * x ≠̇ 0 * ls y 0}
8    t:=x;
9    {∃U. ls x t * t ↦ U * ls U 0 * ls y 0}
10   u:=[t];
11   while {ls x t * t ↦ u * ls u 0 * ls y 0} (u!=0) {
12     {ls x u * ls u 0 * u ≠̇ 0 * ls y 0}
13     t:=u;
14     {∃U. ls x t * t ↦ U * ls U 0 * ls y 0}
15     u:=[t];
16     {ls x t * t ↦ u * ls u 0 * ls y 0}
17   }
18   {ls x t * t ↦ 0 * ls y 0}
19   [t]:=y;
20   {ls x 0}
21 }
22 {ls x 0}
```

Figure 5.2: Proof outline of list append

5.2.1 List append

Figure 5.2 presents a proof outline for an imperative program, previously studied by Berdine et al. [2005], that appends two linked lists. It comprises (rather weak) pre- and postconditions, a loop invariant, and several intermediate assertions to guide the reader through the proof. For a binary relation r, we write $x\ \dot{r}\ y$ for $x\ r\ y \wedge emp$. The ls predicate is the smallest satisfying:

$$ls\ x\ y \quad \Leftrightarrow \quad (x \doteq y \vee x \mathrel{\dot{\neq}} y * \exists x'.\, x \mapsto x' * ls\ x'\ y).$$

Despite the abundance of assertions, the proof outline obscures several features of the proof. For instance, the assertion at the entry to the else-branch (line 7) is potentially confusing because it differs in multiple ways from its predecessor on line 1: '$\mathtt{x} \mathrel{\dot{\neq}} 0$' has appeared, and so has '$ls\ \mathtt{x}\ \mathtt{x}$'. Only the former results from the failure of the test condition; the latter is from a lemma about ls. Likewise, in lines 8 and 13 we perform assignments while expanding the definition of ls, and in line 19 the heap update coincides with the use of an entailment lemma. This common practice of combining multiple proof steps avoids a proliferation of assertions, but comes at the expense of readability. (Displaying each step separately has problems too, as our next example shows.)

In contrast, the corresponding ribbon proof in Fig. 5.3 displays each proof step individually without resorting to repetition. It comprises

- *steps*, each labelled with an instruction (black) or an entailment justification (dark grey), and

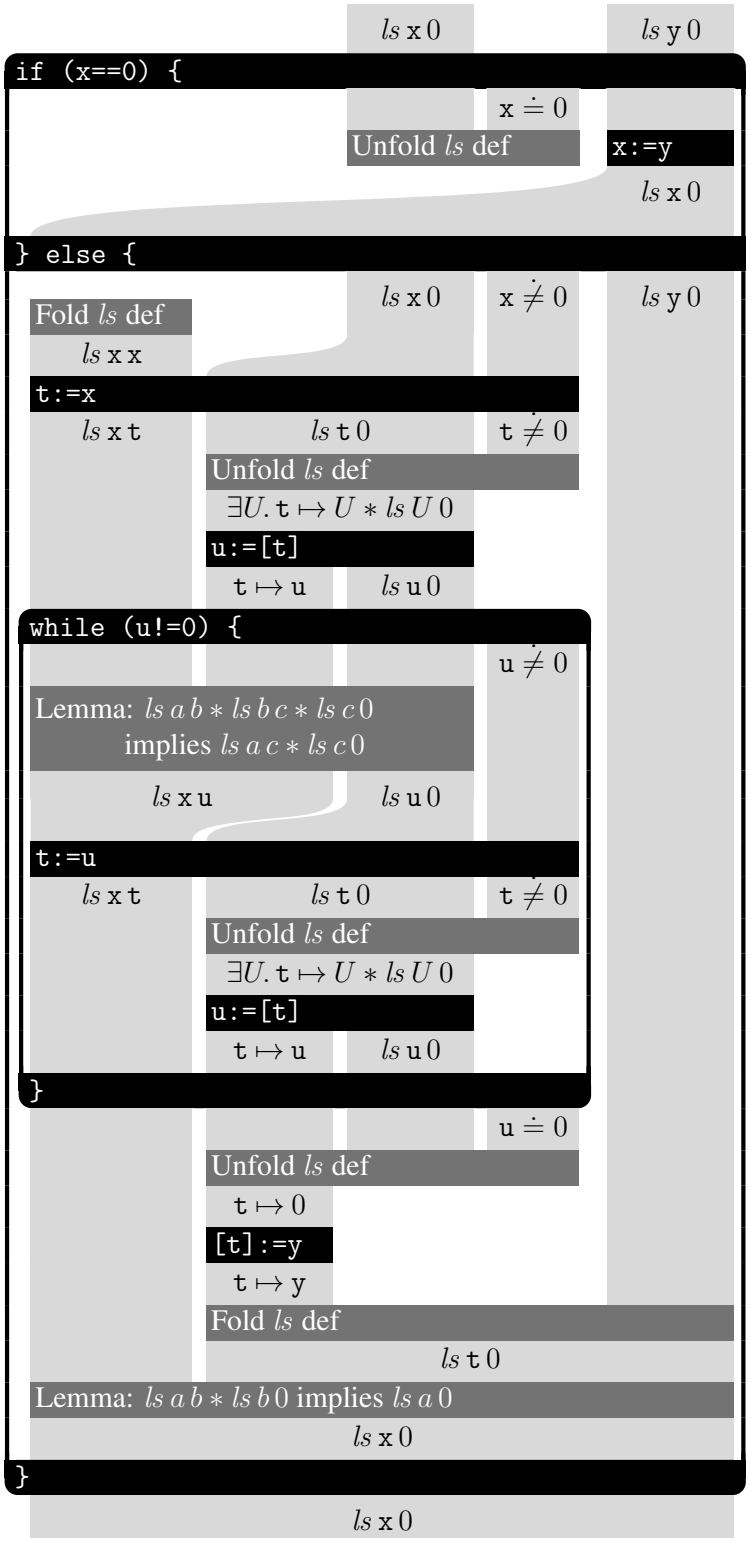


Figure 5.3: Ribbon proof of list append

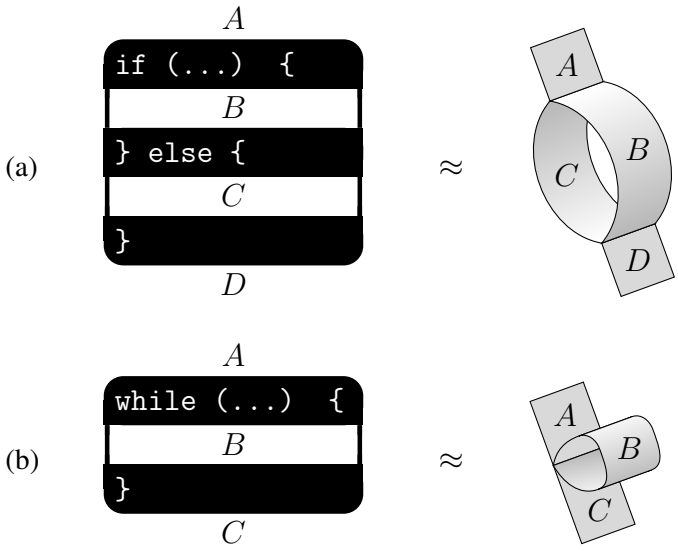


Figure 5.4: If-statements and while-loops, pictorially

- *ribbons* (light grey), each labelled with an assertion.

The ribbon proof advances vertically, and the resources (memory cells) being operated upon are distributed horizontally, across the ribbons. Instructions are positioned according to the resources they access, not merely according to the syntax of the program, as in the proof outline. Horizontal separation between ribbons corresponds to the separating conjunction of the assertions on those ribbons; that is, parallel ribbons refer to disjoint sets of memory cells. Because the $*$-operator is commutative, we can cross one ribbon over another – see Fig. 5.5b for an example of this 'twist' operation. The resource distribution is unordered, and also non-uniform, so the width of a ribbon is not proportional to the amount of resource it describes. In particular, the assertion '$\mathtt{x} \doteq 0$' obtained upon entering the then-branch describes no memory cells at all; it is merely a fact about variables. A gap in the diagram (e.g. above the 'fold' step at the start of the else-branch) corresponds to the 'emp' assertion.

Just above '`t:=u`' we stretch the '$ls\,\mathtt{x}\,\mathtt{u}$' and '$ls\,\mathtt{u}\,0$' ribbons so they align with the corresponding ribbons below the assignment. Such distortions are semantically meaningless but can aid readability. Similarly, at the end of the then-branch we stretch the '$ls\,\mathtt{x}\,0$' ribbon to mimic the ribbon at the end of the else-branch. The general rule is that the collection of ribbons entering the then-branch of an if-statement must match that entering the else-branch, as must the collections at the two exits, so that the proof could be cut and folded into the three-dimensional shape suggested in Fig. 5.4a.

The while-loop has a similar proof structure to the if-statement. Inside the loop body we assume that the test succeeds ($\mathtt{u} \neq 0$); the complementary assumption appears after exiting the loop. The loop invariant is the collection of ribbons entering the top of the loop: $ls\,\mathtt{x}\,\mathtt{t}$ and $\mathtt{t} \mapsto \mathtt{u}$ and $ls\,\mathtt{u}\,0$. This collection must be recreated at the end of the loop body, so that one could roll the proof into the shape drawn in Fig. 5.4b.

In the else-branch, the assertion '$ls\,\mathtt{y}\,0$' is not needed until nearly the end, when it is merged with '$\mathtt{t} \mapsto \mathtt{y}$'. In a proof outline, this assertion would either be temporarily removed via an explicit application of the FRAME rule or, as is done in Fig. 5.2, redundantly repeated at every intermediate point. In the ribbon proof, it slides discreetly down the right-hand column. This indicates that the assertion is *inactive* without suggesting that it has been *removed*.

5.2.2 List reverse

Our second example provides a side-by-side comparison of a proof outline and a ribbon proof, and also explains how ribbon proofs handle existentially-quantified logical variables.

Figure 5.5a gives a proof outline of a program, previously studied by Reynolds [2002], for in-place reversal of a list. We write $\cdot$ for sequence concatenation, $(-)^\dagger$ for sequence reversal and ϵ for the empty sequence, and we define list as the smallest predicate satisfying

$$\begin{aligned} \mathit{list}\,\alpha\,x \;\Leftrightarrow\; & (x \doteq 0 * \alpha \doteq \epsilon) \vee \\ & (x \mathrel{\dot{\neq}} 0 * \exists \alpha', i, x'.\, x \mapsto i, x' * \alpha \doteq i \cdot \alpha' * \mathit{list}\,\alpha'\,x'). \end{aligned}$$

In contrast to Fig. 5.2, this proof outline seeks to clarify the proof by making minimal changes between successive assertions. The cost of this is a large and highly redundant proof. And still the structure of the proof is unclear.

In particular, the proof outline obscures the usage of the logical variables α and β. For instance, the α in line 12 is not the same as the α in line 5, though visually it seems to be. The witness for β is constant through lines 5 to 20, after which it becomes the previous β prepended with i. These subtle changes can only be spotted through careful examination of the proof outline (or else, as we have done, an explicit textual comment). The handling of logical variables in the ribbon proof is far more satisfactory. The scope of a logical variable is delimited by a thin *existential box*. Boxes extend horizontally across several ribbons, but also vertically to indicate the range of steps over which the same witness is used. We are permitted to stretch boxes horizontally – for instance, immediately below the loop in Fig. 5.5b. This corresponds to the implication

$$p * \exists x.\, q \Rightarrow \exists x.\, p * q$$

(where x is not in p). Within any single row projected from the proof, existential boxes must be well-nested; this corresponds to the static scoping of existential quantifiers in assertions. Vertically, however, boxes may overlap; this corresponds to the implication

$$\exists x.\, \exists y.\, p \Rightarrow \exists y.\, \exists x.\, p.$$

Figure 5.6 depicts how the boxes for α and β overlap in Fig. 5.5b. We thus obtain an intriguing proof structure – present in neither the proof outline nor the derivation tree – in which the scopes of logical variables do not follow the program's syntactic structure, but are instead *dynamically* scoped. See Sect. 5.7 for further discussion.

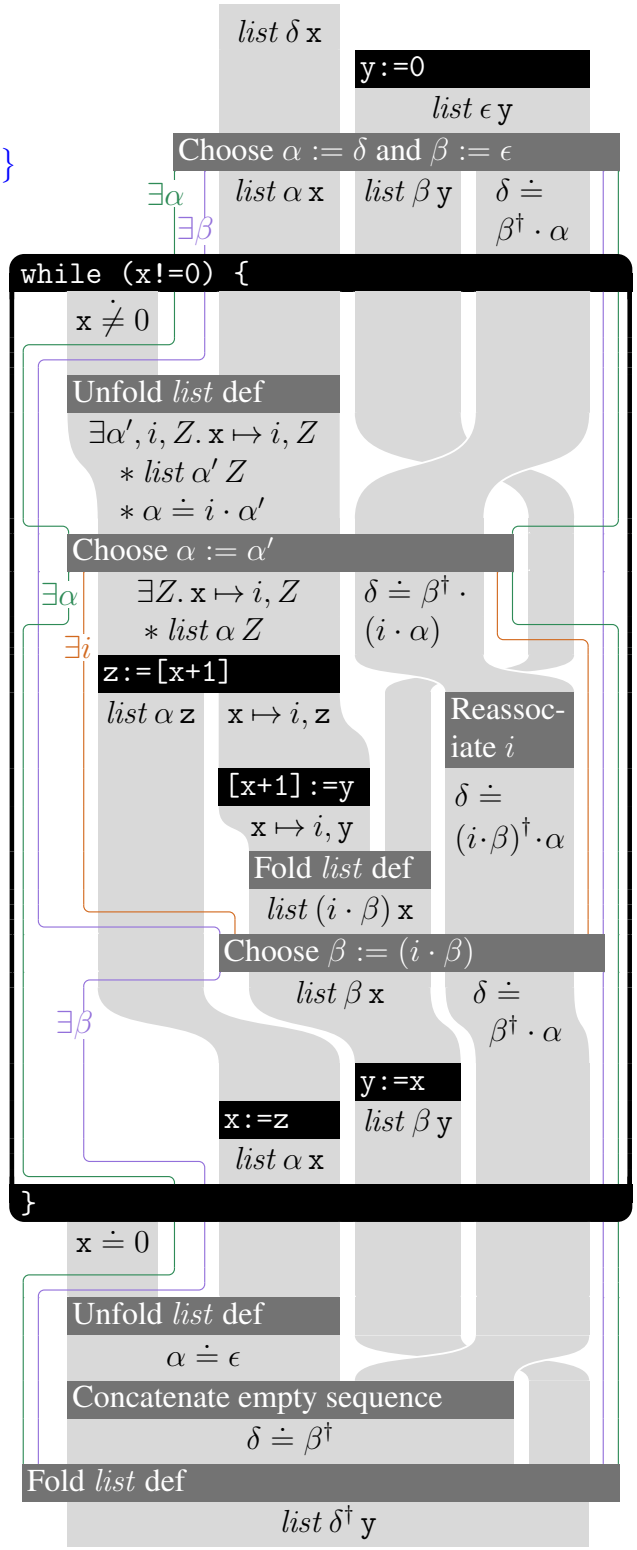
We close this section by explaining a serious shortcoming in the proof system as currently presented. One nicety of Fig. 5.5b is that the 'Reassociate i' entailment is clearly independent of the neighbouring proof steps, being horizontally separated from them, and hence can be safely moved a little earlier or later. Close inspection is necessary to discover this from the proof outline. But by the same reasoning, the assignments '`y:=x`' and '`x:=z`' can be swapped, and this is unsound. This observation will cause difficulties in our formalisation, but we shall overcome them, either by forbidding such manoeuvres altogether (Sect. 5.3) or by embedding information about variable dependencies into the ribbons by using the variables-as-resource paradigm (Sect. 5.4).

5.3 Formalisation

We now formalise the concepts introduced in the previous section. We introduce in Sect. 5.3.1 a two-dimensional syntax for diagrams, and explain how it can generate

1 $\{list\ \delta\ \mathtt{x}\}$
2 `y:=0;`
3 $\{list\ \delta\ \mathtt{x} * list\ \epsilon\ \mathtt{y}\}$
4 `// Choose` $\alpha := \delta$ `and` $\beta := \epsilon$
5 `while` $\{\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha\}$ `(x!=0) {`
6 $\left\{\begin{array}{l}\exists \alpha, \beta.\ \mathtt{x} \neq 0 * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \delta \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
7 `// Unfold` *list* `def`
8 $\left\{\begin{array}{l}\exists \alpha, \beta.\ (\exists \alpha', i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha'\ \mathtt{z}) \\ * \alpha \doteq i \cdot \alpha') * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
9 `// Choose` $\alpha := \alpha'$
10 $\left\{\begin{array}{l}\exists \alpha, \beta, i, Z.\ \mathtt{x} \mapsto i, Z * list\ \alpha\ Z \\ * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$
11 `z:=[x+1];`
12 $\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \delta \doteq \beta^\dagger \cdot (i \cdot \alpha) * list\ \beta\ \mathtt{y}\end{array}\right\}$
13 `// Reassociate` i
14 $\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{z} \\ * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$
15 `[x+1]:=y;`
16 $\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * \mathtt{x} \mapsto i, \mathtt{y} \\ * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha * list\ \beta\ \mathtt{y}\end{array}\right\}$
17 `// Fold` *list* `def`
18 $\left\{\begin{array}{l}\exists \alpha, \beta, i.\ list\ \alpha\ \mathtt{z} * list\ (i \cdot \beta)\ \mathtt{x} \\ * \delta \doteq (i \cdot \beta)^\dagger \cdot \alpha\end{array}\right\}$
19 `// Choose` $\beta := (i \cdot \beta)$
20 $\{\exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{x} * \delta \doteq \beta^\dagger \cdot \alpha\}$
21 `y:=x;`
22 $\{\exists \alpha, \beta.\ list\ \alpha\ \mathtt{z} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha\}$
23 `x:=z;`
24 $\{\exists \alpha, \beta.\ list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha\}$
25 `}`
26 $\left\{\begin{array}{l}\exists \alpha, \beta.\ \mathtt{x} \doteq 0 * list\ \alpha\ \mathtt{x} * list\ \beta\ \mathtt{y} \\ * \delta \doteq \beta^\dagger \cdot \alpha\end{array}\right\}$
27 `// Unfold` *list* `def`
28 $\{\exists \alpha, \beta.\ \alpha \doteq \epsilon * list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger \cdot \alpha\}$
29 `// Concatenate empty sequence`
30 $\{\exists \beta.\ list\ \beta\ \mathtt{y} * \delta \doteq \beta^\dagger\}$
31 `// Fold` *list* `def`
32 $\{list\ \delta^\dagger\ \mathtt{y}\}$

(a) A proof outline



(b) A ribbon proof

Figure 5.5: Two proofs of list reverse

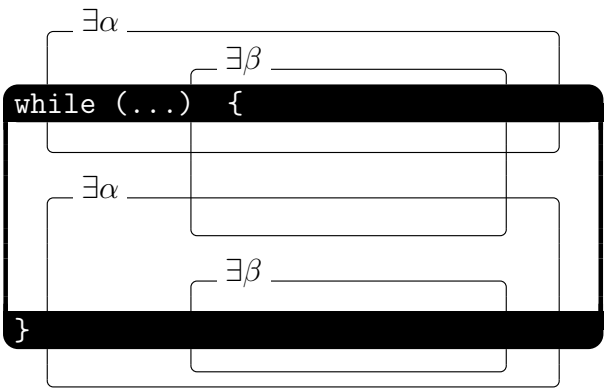


Figure 5.6: Vertical overlapping of existential boxes

the pictures we have already seen. We present the rules of our diagrammatic proof system in Sect. 5.3.2, plus additional rules in Sect. 5.3.3 for composing diagrams in sequence and in parallel. We relate ribbon proofs to separation logic in Sect. 5.3.4.

Proofs performed by hand are annotated with □, while those mechanically verified in Isabelle are annotated with , and can be viewed online at:

`http://www.cl.cam.ac.uk/~jpw48/ribbons.html`

5.3.1 Syntax of ribbon diagrams

We present a syntax that can generate the pictures seen in the preceding section. Each diagram is built up as a sequence of rows, each containing a single proof step. We thus refer to such diagrams as 'stratified'. (Section 5.4 will present an alternative formalisation that does not impose such strict sequentiality.) We begin with the concept of an *interface*, through which diagrams can be connected.

Definition 5.1 (Interfaces)**.** An interface is either a single ribbon labelled with an assertion, an empty interface (shown as whitespace in pictures), two interfaces side by side, or an existential box wrapped around an interface:

$$\mathsf{Interface} \stackrel{\text{def}}{=} \{P ::= \quad p \quad | \quad \varepsilon \quad | \quad P\,P \quad | \quad \exists^{x} P| \ \}.$$

The asn function maps an interface to the assertion it represents:

$$\begin{aligned} asn\ p &= p \\ asn\,\varepsilon &= emp \\ asn\,(P\,Q) &= asn\,P * asn\,Q \\ asn\ \exists^{x} P| &= \exists x.\ asn\,P. \end{aligned}$$

Where clarity demands it, we shall write $P \otimes Q$ instead of $P\,Q$, and hence $\otimes_{i\in I} P_i$ for iterated composition. Interfaces are identified up to $(P\,Q)\,R = P\,(Q\,R)$ and $P\,\varepsilon = \varepsilon\,P = P$ and $P\,Q = Q\,P$. By making $\otimes$ commutative, the 'twisting' of ribbons becomes merely a presentational issue.

A *diagram* can be thought of as a mapping between two interfaces.

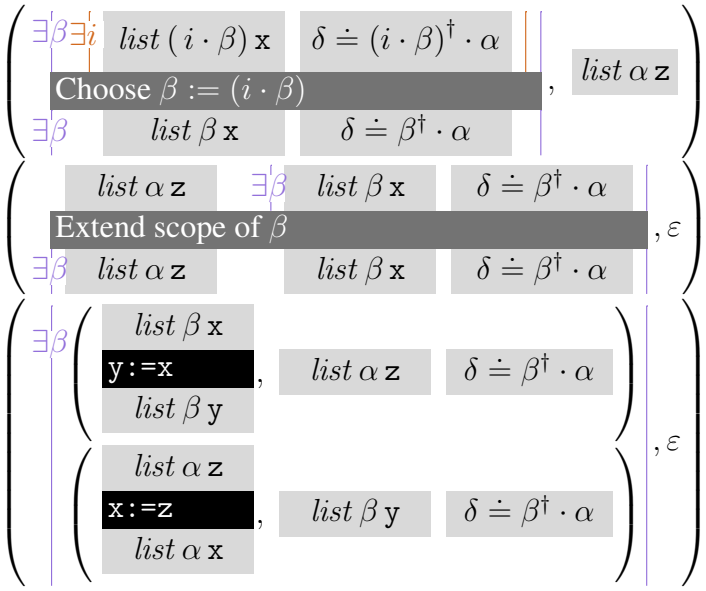


Figure 5.7: Stratified parsing of a fragment of Fig. 5.5b

Definition 5.2 (Diagrams)**.** A diagram $D \in \mathsf{Diagram}$ is a non-empty list of *rows* $\rho \in \mathsf{Row}$. The list $[\rho_0, \ldots, \rho_k]$ is alternatively written as

$$\begin{array}{c} \rho_0 \\ \vdots \\ \rho_k \end{array}$$

when space permits. A row is a pair (γ, F) comprising a *cell* $\gamma \in \mathsf{Cell}$ and a *frame* $F \in \mathsf{Interface}$. The syntax of cells is as follows:

$$\mathsf{Cell} \stackrel{\text{def}}{=} \{\gamma ::= P \mid \begin{array}{c} P \\ c \\ P \end{array} \mid {}^{\exists x}_{\dagger} D \mid \begin{array}{c} P \\ D \\ \texttt{or} \\ D \\ P \end{array} \mid \begin{array}{c} P \\ \texttt{loop} \\ P \\ P \end{array} \}.$$

To illustrate how this syntax is used, Fig. 5.7 shows a term of Diagram that corresponds to a fragment of the picture in Fig. 5.5b. Note that the cell in each row is always pushed to the left-hand side. In the concrete pictures, the cell can be moved to allow corresponding ribbons in different rows to be aligned, and hence for redundant labels to be removed. Each entailment $p \Rightarrow q$ is handled as the basic step $\{p\}$ `skip` $\{q\}$. Rather than write '`skip`', we label such a step with a justification of the entailment, and colour it dark grey to emphasise those steps that actually contain program instructions. Concerning existential boxes: the operations of extending, contracting and commuting are really the entailments depicted informally in Fig. 5.8. Having to show these entailments explicitly would make Fig. 5.5b much more repetitive. We are working on an improved formalisation that supports these operations directly – see Sect. 5.7 for further discussion.

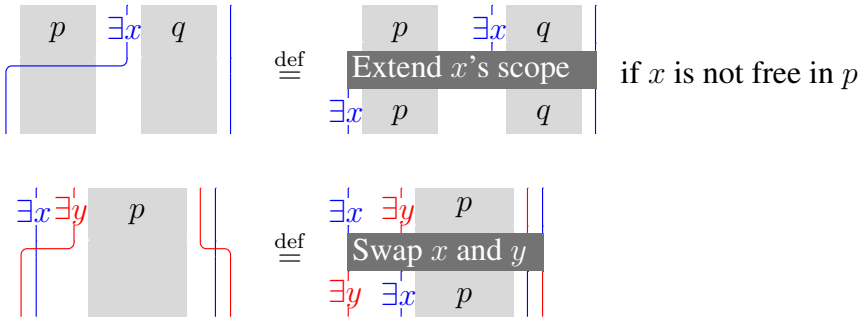


Figure 5.8: Syntactic sugar for existential boxes

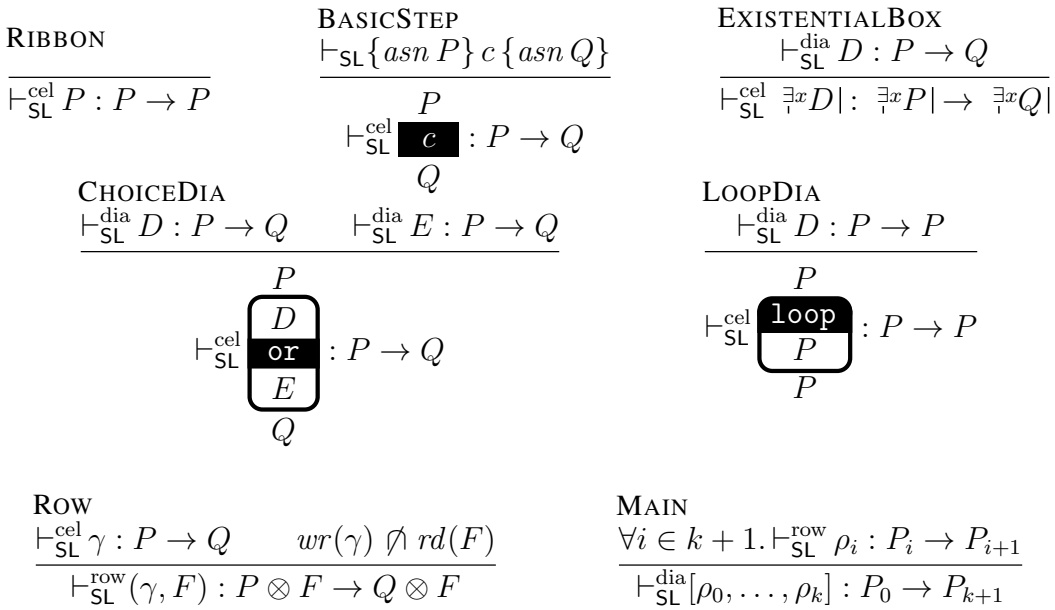


Figure 5.9: Proof rules for stratified ribbon diagrams

5.3.2 Proof rules for diagrams

There are two pertinent questions to be asked of a given ribbon diagram. The first question is: is it a valid proof? This subsection develops a *provability* judgement to answer this. The second question – if this ribbon diagram *is* deemed valid, what does it prove? – is addressed in Section 5.3.4.

The rules given in Fig. 5.9 define provability judgements for cells ($\vdash^{\text{cel}}_{\mathsf{SL}}$), for rows ($\vdash^{\text{row}}_{\mathsf{SL}}$) and for diagrams ($\vdash^{\text{dia}}_{\mathsf{SL}}$). Each judgement assigns a type, which comprises the top and bottom interfaces of that object.

The MAIN rule recalls an iterated version of Hoare logic's sequencing rule, while the ROW rule recalls separation logic's FRAME rule. Together, these rules embody the 'locally checkable' nature of ribbon proofs: that an entire diagram is valid if each row is valid in isolation, and that a row is valid if its cell is valid and writes no program variable that is read elsewhere in the row.

The BASICSTEP rule corresponds to an ordinary separation logic judgement $\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\}$. This judgement may be arbitrarily complex, so a ribbon diagram may be no easier to check than a traditional proof outline. This is intentional. Our formalisation *allows* p and q to be minimised, by framing common fragments away, but does not *demand* this. The command

c *can* be reduced to `skip` or some primitive command, but there are times where this is not desirable; for instance, if one requires only a high-level overview proof. A ribbon diagram can thus be viewed as a flexible combination of diagrammatic and traditional proofs, with the BASICSTEP rule as the interface between the two levels.

5.3.3 Composition of diagrams

We remark that the proof rules presented in the previous section provide only limited mechanisms for building new diagrams from old. Diagrams can be wrapped in existential boxes, put inside choice or loop diagrams, but not stacked vertically or placed side by side. In this subsection, we derive two additional proof rules for composing stratified diagrams in sequence or in parallel.

Sequential (vertical) composition is a straightforward matter of list concatenation, which we write as $\frac{D}{E}$ for diagrams D and E. Parallel (horizontal) composition is a little fiddly: because our diagrams do not carry absolute positional information, we must state explicitly how to interleave the proof steps.

Definition 5.3 (Top and bottom interfaces of stratified diagrams)**.** The following mutually recursive function extracts the top interface from a cell, a row, or a diagram. (The *bot* function is defined similarly.)

$$top[(\gamma_0, F_0), \dots] \;=\; top(\gamma_0) \otimes F_0 \qquad top(P) \;=\; P \qquad top\ {}^{\exists x}_{\mid} D \mid \;=\; {}^{\exists x}_{\mid} top\ D \mid$$

$$top \begin{array}{c} P \\ \hline c \\ \hline Q \end{array} \;=\; P \qquad top \begin{array}{|c|} \hline P \\ \hline D \\ \texttt{or} \\ E \\ \hline Q \\ \hline \end{array} \;=\; P \qquad top \begin{array}{|c|} \hline P \\ \texttt{loop} \\ P \\ \hline Q \\ \hline \end{array} \;=\; P$$

Definition 5.4 (Parallel composition of stratified diagrams)**.** If D and E are diagrams of lengths m and n, and μ is a binary sequence containing m zeroes and n ones, then we define the parallel composition of D and E according to μ as follows:

$$D \parallel_\mu E \;\stackrel{\text{def}}{=}\; zip_\mu(D, E, top\ D, top\ E)$$

where *zip* is defined inductively as follows:

$$\begin{aligned} zip_\epsilon(\_, \_, \_, \_) &= [\,] \\ zip_{0\mu}((\gamma, F) :: D, E, \_, Q) &= (\gamma, F \otimes Q) :: zip_\mu(D, E, bot\,\gamma \otimes F, Q) \\ zip_{1\mu}(D, (\gamma, F) :: E, P, \_) &= (\gamma, P \otimes F) :: zip_\mu(D, E, P, bot\,\gamma \otimes F). \end{aligned}$$

One way to obtain the ribbon diagram in Fig. 5.1b is to compose its first two columns in parallel with its third, as shown in Fig. 5.10.

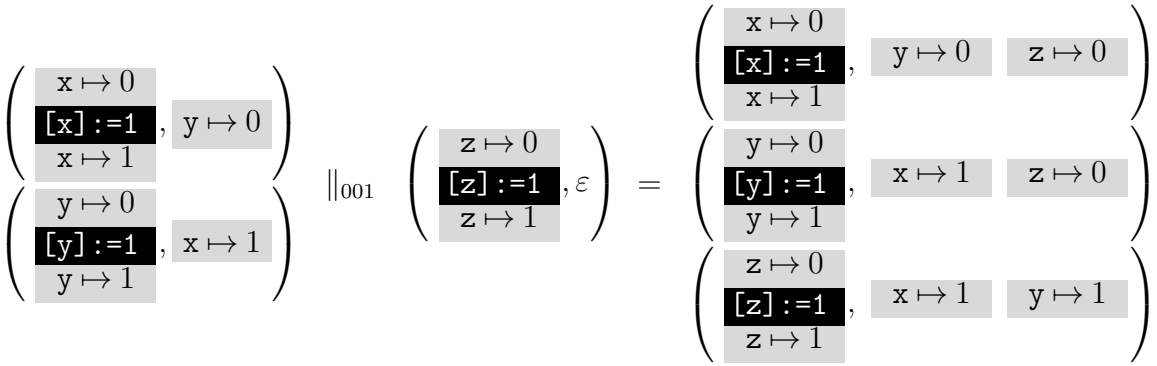


Figure 5.10: Parallel composition of stratified diagrams, an example

$$com[(\gamma_0, F_0), \ldots, (\gamma_k, F_k)] = com\,\gamma_0 \,;\, \cdots \,;\, com\,\gamma_k \qquad com\,P = \texttt{skip} \qquad com\ {}^{\exists x}_{\mid} D \mid = com\,D$$

$$com\begin{array}{c} P \\ \boxed{c} \\ Q \end{array} = c \qquad com\begin{array}{c} P \\ \boxed{\texttt{loop}\ P} \\ Q \end{array} = \texttt{loop}(com\,D) \qquad com\begin{array}{c} P \\ \boxed{\begin{array}{c} D \\ \texttt{or} \\ E \end{array}} \\ Q \end{array} = \begin{array}{l}(com\,D) \\ \texttt{or}(com\,E)\end{array}$$

Figure 5.11: Extracting a command from a stratified diagram

Theorem 5.5. *The following rules are derivable from those in Fig. 5.9.*

$$\textsc{SeqComp}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : Q \to R}{\vdash^{\text{rdia}}_{\mathsf{SL}} \begin{array}{c} D \\ E \end{array} : P \to R} \qquad \textsc{ParComp}\ \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to Q' \qquad D \,\#\, E}{\vdash^{\text{rdia}}_{\mathsf{SL}} D \parallel_{\mu} E : P\,P' \to Q\,Q'}$$

Proof. See Appx. A.1. □

5.3.4 Semantics of diagrams

A stratified ribbon diagram denotes a Hoare triple. The pre- and postconditions of this triple are the assertions represented by the diagram's top and bottom interfaces. The command being proved is extracted by composing the labels on all of the proof steps in top-to-bottom order. Figure 5.11 defines the function responsible for this extraction. We can now phrase the following soundness result for ribbon proofs.

Theorem 5.6 (Soundness – stratified diagrams)**.** *Separation logic can encode any provable ribbon diagram.*

$$\vdash^{\text{dia}}_{\mathsf{SL}} D : P \to Q \implies \vdash_{\mathsf{SL}} \{asn\,P\}\ com\,D\,\{asn\,Q\}.$$

Proof. By mutual rule induction on $\vdash^{\text{cel}}_{\mathsf{SL}}$, $\vdash^{\text{row}}_{\mathsf{SL}}$, and $\vdash^{\text{dia}}_{\mathsf{SL}}$. 🍒

Concerning completeness: ribbon diagrams are trivially complete, because the BASICSTEP rule can be invoked right at the root of the proof tree. That is, any separation logic judgement can

be written directly as a ribbon proof comprising a single basic step. But do ribbon diagrams remain complete even when the BASICSTEP rule can occur only immediately beneath an axiom or the rule of consequence? They do, providing the Hoare Logic rule of conjunction (CONJ) is discarded, there being no analogue for that rule in our ribbon proof system. We have thus far presented the rules of separation logic as being open-ended, but to phrase our completeness result, we must restrict the set of separation logic rules to those that can be mimicked by ribbon proof rules, that is: FRAME, EXISTS, DISJ, CONSEQ, CHOICE, SEQ, SKIP, LOOP, and a set Axioms of axioms.

Theorem 5.7 (Completeness – stratified diagrams)**.** *A strengthened ribbon proof system in which the* BASICSTEP *rule is replaced by*

$$\frac{(asn\,P, c, asn\,Q) \in \mathsf{Axioms}}{\vdash^{\mathrm{cel}}_{\mathsf{SL}} \begin{array}{c} P \\ \boxed{c} \\ Q \end{array} : P \to Q} \qquad \frac{asn\,P \Rightarrow asn\,Q}{\vdash^{\mathrm{cel}}_{\mathsf{SL}} \begin{array}{c} P \\ \boxed{\texttt{skip}} \\ Q \end{array} : P \to Q}$$

can encode any separation logic proof.

$$\vdash_{\mathsf{SL}} \{p\}\, c\, \{q\} \implies \exists D, P, Q.\, c \in com\, D \wedge p = asn\, P \wedge q = asn\, Q \wedge \vdash^{\mathrm{dia}}_{\mathsf{SL}} D : P \to Q$$

Proof. By rule induction on $\vdash_{\mathsf{SL}}$. □ □

The main problem with the formalisation given in this section is that it sacrifices much of the flexibility we expect in our ribbon diagrams. It is often sound to tweak the layout of a diagram by sliding steps up or down or by reordering ribbons, but by thinking of our diagrams as sliced into a sequence of rows, we rule out *all* such manoeuvres.

5.4 Graphical formalisation

We now give an alternative formalisation, in which diagrams are represented not as a sequence of rows, but as graphs. These 'graphical' diagrams are more flexible than their 'stratified' cousins, but extra precautions must be taken to ensure soundness. The obstacle to soundness is the side-condition on the FRAME rule: that the command writes no program variable in the frame. With stratification, the frame is clearly delimited and the condition easily checked. Without it, the check becomes more global: a command may affect a ribbon that appears, in the laid-out diagram, far above or below it. We wish to eliminate this side-condition altogether, and one way to do so is to use variables-as-resource [Bornat et al. 2006].

The graphs we use are nested, directed, acyclic hypergraphs. Ribbons correspond to nodes, and basic steps correspond to hyperedges. Existential boxes are represented as single nodes that containing a nested graph. Likewise, choice diagrams and loop diagrams are represented by single hyperedges that contain, respectively, one or two nested graphs.

Regarding notation: we tend to use letters in upper case or bold type to range over sets. We sometimes treat a natural number k as the ordinal $\{0, \ldots, k-1\}$. If R is a relation, then let R^0 be the identity relation, and R^{n+1} be the composition of R with R^n. Let $\mathcal{V}$ be an infinite set of node-identifiers.

Definition 5.8 (Graphical diagrams, assertion-gadgets and command-gadgets)**.** The equations below define a language of *graphical diagrams*, *assertion-gadgets* and *command-gadgets*.

$$\mathsf{AsnGadget} = \{A ::= p \mid \exists x\ G\}$$

$$\mathsf{ComGadget} = \{\chi ::= c \mid \begin{array}{|c|}\hline G \\ \texttt{or} \\ G \\ \hline\end{array} \mid \begin{array}{|c|}\hline \texttt{loop} \\ G \\ \hline\end{array}\}$$

$$\mathsf{GDiagram} = \{G \mid \Lambda_G \in V_G \to \mathsf{AsnGadget},\ E_G \subseteq_{\text{fin}} \mathcal{P}(V_G) \times \mathsf{ComGadget} \times \mathcal{P}(V_G),\ V_G \subseteq_{\text{fin}} \mathcal{V},\ \mathit{acyclic}(G) \text{ and } \mathit{linear}(G), \text{ where } G = (V_G, \Lambda_G, E_G)\}$$

The definitions are mutually recursive, and are well-formed because the definienda (left-hand sides) appear only positively in the definientia (right-hand sides). (This is true even for the occurrence of ComGadget in the definiens of GDiagram, because the set in which it appears is finite.) The first of these equations defines an assertion-gadget $A \in \mathsf{AsnGadget}$ to be either a ribbon or an existential box. The second defines a command-gadget $\chi \in \mathsf{ComGadget}$ to be either a basic step, a choice diagram, or a loop diagram. The third equation defines a graphical diagram $G \in \mathsf{GDiagram}$ to be a triple (V_G, Λ_G, E_G) that comprises:

- a finite set $V_G \subseteq_{\text{fin}} \mathcal{V}$ of *node identifiers*;
- a *labelling* $\Lambda_G : V_G \to \mathsf{AsnGadget}$ that associates each node identifier with an assertion-gadget; and
- a finite set $E_G \subseteq_{\text{fin}} \mathcal{P}(V_G) \times \mathsf{ComGadget} \times \mathcal{P}(V_G)$ of *hyperedges* $(\mathbf{v}, \chi, \mathbf{w})$, each comprising a set $\mathbf{v}$ of tail identifiers, a command-gadget χ, and a set $\mathbf{w}$ of head identifiers,

and which satisfies the following two properties.

ACYCLICITY: Let us write $v \rightarrowtail w$ if $v \in \mathbf{v}$ and $w \in \mathbf{w}$ for some $(\mathbf{v}, \chi, \mathbf{w}) \in E_G$. Then $\mathit{acyclic}(G)$ iff $v \rightarrowtail^i v$ implies $i = 0$.

LINEARITY: Define $\mathit{linear}(G)$ to hold iff the hyperedges in E_G have no common heads and no common tails.

Linearity models the fact that ribbons cannot be duplicated, which in turn is a result of $p \Rightarrow p * p$ being invalid in separation logic.

We remark that we could represent our diagrams by a single graph, with dedicated 'parent' edges to simulate the nesting hierarchy. However, mindful of our Isabelle formalisation, and that "reasoning about graphs [...] can be a real hassle in HOL-based theorem provers" [Wu et al. 2011], we prefer to use an inductive datatype to depict the hierarchy. The steps and ribbons at each level of the hierarchy, however, do not form a tree structure, so must remain non-inductive.

Remark 5.9. We usually work with *abstract diagrams*. These diagrams are identified up to graph isomorphism; that is, the particular choice of node-identifiers is unimportant. In particular, the diagrams that appear within assertion-gadgets or command-gadgets are treated abstractly. However, some definitions and proofs work with *concrete diagrams* where the node-identifiers are exposed.

Figure 5.12 presents the term of GDiagram that corresponds to the picture in Fig. 5.5b. Note that we display the entailment steps defined in Fig. 5.8 like so: ■. Unlike Fig. 5.7, this representation

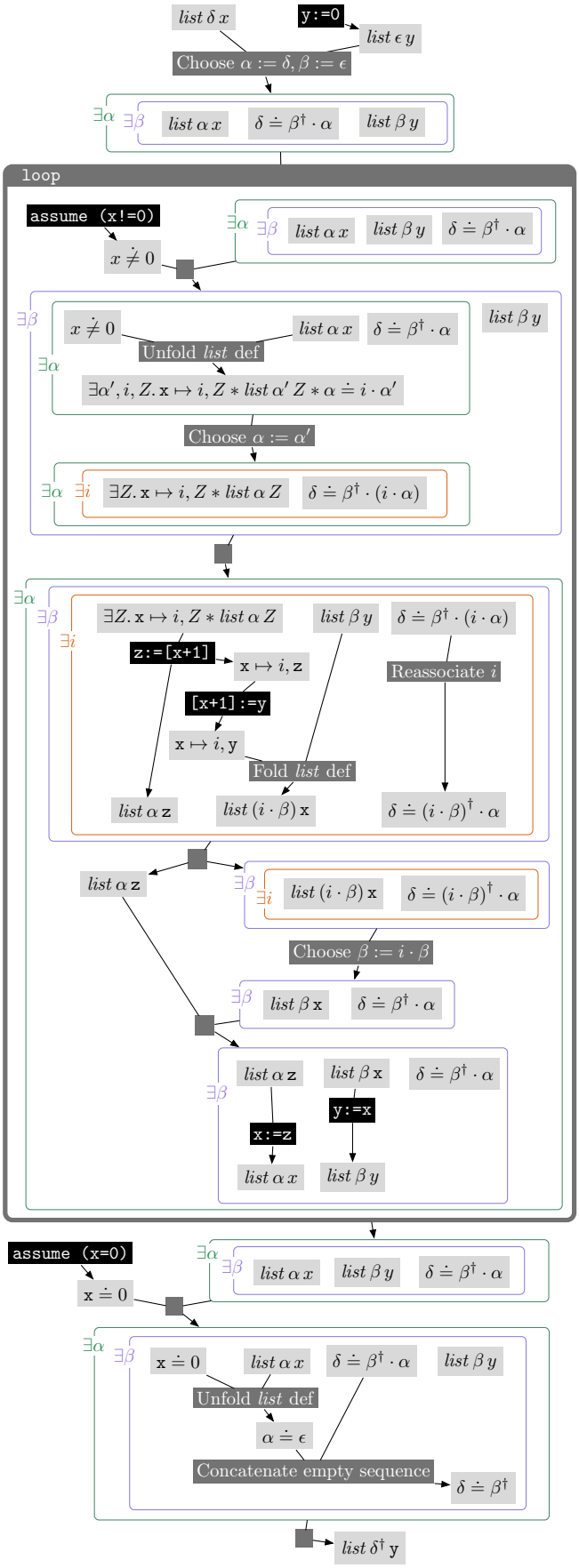


Figure 5.12: Graphical parsing of Fig. 5.5b

does not impose a strict ordering between the '`y:=x`' and '`x:=z`' instructions. Because these instructions cannot safely be permuted, the proof, when represented in this way, is invalid. The problem is that the graph does not take into account dependencies on program variables. To attain a sound proof system for graphical diagrams, we turn to the *variables-as-resource* paradigm.

As explained in Sect. 2.5.3, the variables-as-resource paradigm treats program variables a little like separation logic treats heap cells. Each program variable x is associated with a piece of resource, all of which (written $Own_1(x)$) must be held to write to x, and some of which ($Own_\pi(x)$ for some $0 < \pi \leq 1$) must be held to read it. Thanks to the generality of our formalisation, we can obtain the variables-as-resource proof system (written $\vdash_{\mathsf{VaR}}$) simply by removing the side-condition on the FRAME rule and selecting an appropriate set of axioms.

5.4.1 Proof rules for graphical diagrams

Proof rules for graphical diagrams, command-gadgets and assertion-gadgets are defined in Fig. 5.13, which refers to the top and bot functions defined below. The judgement

$$\vdash^{\mathrm{gra}}_{\mathsf{VaR}} G : P \to Q$$

means that the diagram G, precondition P, and postcondition Q form a valid proof. The interfaces P and Q are always equal to $top(G)$ and $bot(G)$ respectively, so we sometimes omit them. The judgements for command-gadgets and assertion-gadgets are similar, the latter without interfaces.

Definition 5.10 (Top and bottom interfaces of graphical diagrams)**.** These functions extract interfaces from assertion-gadgets and from diagrams. For assertion-gadgets:

$$top\ \boxed{p} = \boxed{p} \qquad bot\ \boxed{p} = \boxed{p} \qquad top\ \boxed{\exists x\, G} = \overset{\exists x}{\restriction} top\, G| \qquad bot\ \boxed{\exists x\, G} = \overset{\exists x}{\restriction} bot\, G|.$$

For diagrams:

$$top(G) = \otimes_{v \in initials\, G}\ top(\Lambda_G\, v) \qquad bot(G) = \otimes_{v \in terminals\, G}\ bot(\Lambda_G\, v)$$

where

$$initials\, G = V_G \setminus \bigcup\{\mathbf{v} \mid (\_, \_, \mathbf{v}) \in E_G\} \qquad terminals\, G = V_G \setminus \bigcup\{\mathbf{v} \mid (\mathbf{v}, \_, \_) \in E_G\}.$$

5.4.2 Composition of graphical diagrams

For composing graphical diagrams, it is possible to derive proof rules in the spirit of those in Thm. 5.5.

Definition 5.11 (Sequential composition of graphical diagrams)**.** We notate sequential composition by vertical stacking. We overload this notation for both diagrams and assertion-gadgets. If G and H are diagrams for which:

- $terminals\, G = initials\, H = V_G \cap V_H$, and

$$\textsc{GRibbon}\ \frac{}{\vdash^{\text{asn}}_{\mathsf{VaR}} p} \qquad \textsc{GBasicStep}\ \frac{\vdash_{\mathsf{VaR}} \{asn\, P\}\, c\, \{asn\, Q\}}{\vdash^{\text{com}}_{\mathsf{VaR}} \boxed{c} : P \to Q} \qquad \textsc{GExistentialBox}\ \frac{\vdash^{\text{gra}}_{\mathsf{VaR}} G}{\vdash^{\text{asn}}_{\mathsf{VaR}} \boxed{\exists x\, G}}$$

$$\textsc{GChoiceDia}\ \frac{\vdash^{\text{gra}}_{\mathsf{VaR}} G_1 : P \to Q \quad \vdash^{\text{gra}}_{\mathsf{VaR}} G_2 : P \to Q}{\vdash^{\text{com}}_{\mathsf{VaR}} \begin{array}{|c|}\hline G_1 \\ \texttt{or} \\ G_2 \\ \hline \end{array} : P \to Q}$$

$$\textsc{GLoopDia}\ \frac{\vdash^{\text{gra}}_{\mathsf{VaR}} G : P \to P}{\vdash^{\text{com}}_{\mathsf{VaR}} \begin{array}{|c|}\hline \texttt{loop} \\ G \\ \hline \end{array} : P \to P}$$

$$\textsc{GMain}\ \frac{\forall v \in V_G.\, \vdash^{\text{asn}}_{\mathsf{VaR}} \Lambda_G\, v \qquad \forall (\mathbf{v}, \chi, \mathbf{w}) \in E_G.\, \vdash^{\text{com}}_{\mathsf{VaR}} \chi : \otimes_{v \in \mathbf{v}}\, bot(\Lambda_G\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda_G\, w)}{\vdash^{\text{gra}}_{\mathsf{VaR}} G : top(G) \to bot(G)}$$

Figure 5.13: Proof rules for graphical diagrams

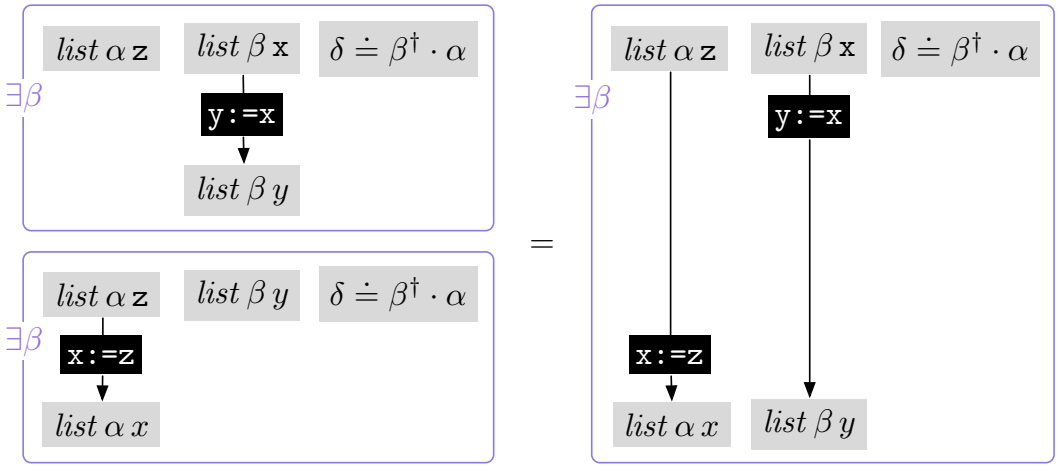


Figure 5.14: Sequential composition of graphical diagrams, an example

- $\begin{matrix}\Lambda_G(v)\\ \Lambda_H(v)\end{matrix}$ is defined for all $v \in V_G \cap V_H$

then we write $\begin{matrix}G\\ H\end{matrix}$ for the diagram $(V_G \cup V_H, \Lambda, E_G \cup E_H)$, where

$$\Lambda(v) = \begin{cases} \Lambda_G(v) & \text{if } v \in V_G \setminus V_H \\ \Lambda_H(v) & \text{if } v \in V_H \setminus V_G \\ \begin{pmatrix}\Lambda_G(v)\\ \Lambda_H(v)\end{pmatrix} & \text{if } v \in V_G \cap V_H. \end{cases}$$

Simultaneously, sequential composition on assertion-gadgets is (partially) defined as follows:

$$\begin{matrix}p\\ p\end{matrix} = p \qquad \begin{matrix}\boxed{\exists x\, G}\\ \boxed{\exists x\, H}\end{matrix} = \boxed{\exists x \begin{matrix}G\\ H\end{matrix}} \text{ provided } \begin{pmatrix}G\\ H\end{pmatrix} \text{ is defined.}$$

The definition above appears fiddly, but it becomes natural once the diagrams are drawn. Fig-

ure 5.14 shows how two diagrams – each comprising a single existential box around four ribbons and one basic step – can be sequentially composed.

Definition 5.12 (Parallel composition of graphical diagrams)**.** If G and H are diagrams with disjoint sets of node-identifiers, then we write $G \parallel H$ for the diagram

$$(V_G \cup V_H, \Lambda_G \cup \Lambda_H, E_G \cup E_H).$$

Theorem 5.13. *The following rules are derivable from those in Fig. 5.13.*

$$\textsc{GSeqComp}\quad \frac{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q \qquad \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : Q \to R}{\vdash^{\mathrm{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix} : P \to R}$$

$$\textsc{GParComp}\quad \frac{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q \qquad \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : P' \to Q'}{\vdash^{\mathrm{dia}}_{\mathsf{SL}} G \parallel H : P\,P' \to Q\,Q'}$$

Proof. See Appx. A.2. □

5.4.3 Semantics of graphical diagrams

Since graphical diagrams have a parallel nature, but our language is only sequential, it follows that each graphical diagram proves not a single command, but a set of commands, each one a linear extension of the partial order imposed by the diagram. The $coms$ function defined in Fig. 5.15 is responsible for extracting this set from a given diagram. Each command is obtained by picking an ordering of command- and assertion-gadgets that is compatible with the partial order defined by the edges (this is the purpose of the lin function defined below), then recursively extracting a command from each gadget and sequentially composing the results.

Definition 5.14 (Linear extensions)**.** For a diagram G, we define $lin\, G$ as the set of all lists $[x_0, \ldots, x_{k-1}]$ of AsnGadgets and ComGadgets, for which there exists a bijection $\pi : k \to V_G \uplus E_G$ that satisfies, for all $(\mathbf{v}, \chi, \mathbf{w}) \in E_G$:

$$\forall v \in \mathbf{v}.\, \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) \qquad \forall w \in \mathbf{w}.\, \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) < \pi^{-1}(w)$$

and where, for all $i \in k$:

$$x_i = \begin{cases} \Lambda_G(v) & \text{if } \pi(i) = v \\ \chi & \text{if } \pi(i) = (\mathbf{v}, \chi, \mathbf{w}). \end{cases}$$

By ACYCLICITY, every diagram admits at least one linear extension.

Theorem 5.15 (Soundness – graphical diagrams)**.** *Separation logic with variables-as-resource can encode any ribbon diagram that is provable with variables-as-resource:*

$$\vdash^{\mathrm{gra}}_{\mathsf{VaR}} G : P \to Q \implies \forall c \in coms\, G. \vdash_{\mathsf{VaR}} \{asn\, P\}\, c\, \{asn\, Q\}.$$

Proof. See Appx. A.3. ☘

$$coms(G) \;=\; \{c_0 \,;\, \cdots \,;\, c_{k-1} \,;\, \texttt{skip} \mid \exists [x_0, \ldots, x_{k-1}] \in lin\, G.\, \forall i \in k.\, c_i \in coms\, x_i\}$$

$$coms\ \boxed{p} \;=\; \{\texttt{skip}\} \qquad coms\ \boxed{\exists x\, G} \;=\; coms\, G \qquad coms\ \boxed{c} \;=\; \{c\}$$

$$coms\ \boxed{\begin{array}{c} G_1 \\ \texttt{or} \\ G_2 \end{array}} \;=\; \begin{array}{l} \{c_1 \texttt{ or } c_2 \mid \\ \quad c_1 \in coms\, G_1, \\ \quad c_2 \in coms\, G_2\} \end{array} \qquad coms\ \boxed{\begin{array}{c} \texttt{loop} \\ G \end{array}} \;=\; \{\texttt{loop}\, c \mid c \in coms\, G\}$$

Figure 5.15: Extracting commands from a graphical diagram

Figure 5.16 exhibits a ribbon proof, conducted using variables-as-resource, of the list-reversal program from §5.2.2. Variables-as-resource dictates that every assertion in the proof is well-scoped; that is, accompanied by one *Own* predicate for each program variable it mentions. For instance, the precondition $list\, \delta\, \texttt{x}$ is paired with some of x's resource. The shading is merely syntactic sugar; for instance:

$$\texttt{x}, \tfrac{1}{2}\texttt{y}\ \ \texttt{x} \mapsto i, \texttt{y} \quad \stackrel{\text{def}}{=} \quad Own_1(\texttt{x}) * Own_{.5}(\texttt{y}) * \texttt{x} \mapsto i, \texttt{y}\,.$$

The other preconditions – the resources associated with y and z – entitle the program to write to these program variables in due course. Note that at the entry to the while loop, part of x's resource is required in order to carry out the test of whether x is zero. At various points in the proof, variable resources are split or combined, but their total is always conserved. Figure 5.16 introduces a couple of novel features: ribbons may pass 'underneath' basic steps to reduce the need for twisting (see e.g. the 'Choose $\alpha := \delta$ and $\beta := \epsilon$' step), and horizontal space is conserved by writing some assertions sideways. The diagram can be laid out in several ways, unconstrained by the stratification strategy of the previous section, so there exists the potential to use the same diagram to justify several variations of a program. Recall the shortcoming of Fig. 5.5b, that it misleadingly suggested that '`y:=x`' and '`x:=z`' could be safely permuted. Figure 5.16 forbids this, by showing the dependency on '`x`' explicitly as a ribbon between the steps. On the other hand, both figures agree that the 'Reassociate i' step can be safely manoeuvred up or down a little.

In this section and the previous one, we have presented two alternative formalisations of ribbon diagrams. We remark that one who seeks merely to present a proof of a particular program need not use variables-as-resource; the splitting, distributing, and re-combining of the resource associated with each variable is an unnecessary burden. Figure 5.16 is significantly larger and fiddlier than Fig. 5.5b, which does not use variables-as-resource. Concrete pictures should be drawn carefully so they can be successfully sliced into rows. Conversely, one who seeks to explore potential optimisations, or to analyse the dependencies between various components of a program, should invest in variables-as-resource.

5.5 Ribbon proof of Version 7 Unix memory manager

In this section, we illustrate the ability of our system to produce readable proofs for more complex programs. As a case study, we revisit the memory manager from Version 7 Unix, for which a proof outline was given in Sect. 4.3.8.

A ribbon proof for the Version 7 Unix memory manager is too large to be covered in full detail all at once. Fortunately, our ribbon diagrams support hierarchical proof construction.

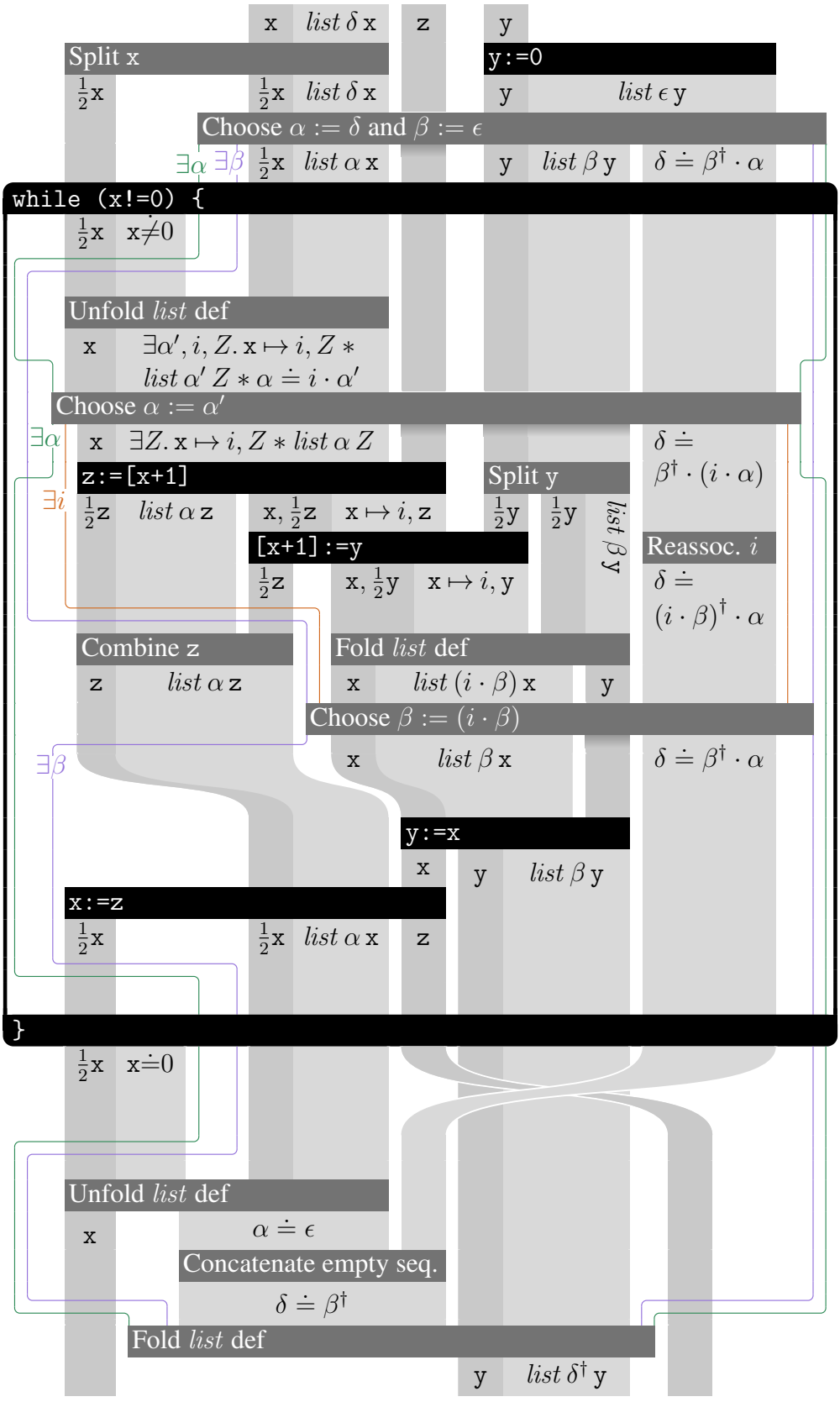


Figure 5.16: Ribbon proof of list reverse using variables-as-resource

Future work will provide tool support for exploring such hierarchical proofs, but until then, we are limited to a static presentation. We provide an overview of the entire `malloc` routine in Fig. 5.17. Figures 5.18 to 5.22 expand each of the main phases, each of which is depicted as a single basic step in the overview proof. Figure 5.23 expands one of the steps of Fig. 5.20 in high detail, to illustrate the range of granularities supported by ribbon proofs.

The ribbon proof is best read by concentrating on each command c in turn, and checking that it correctly transforms those ribbons directly above it into those directly below it. Ribbons to the left or right of c can largely be ignored; the only requirement upon them is not to mention any program variable that c writes.

Because the implementation uses unstructured control flow, several commands are wider than may seem necessary. For instance, the $brka$ predicate must be passed all the way through the main for-loop because the loop contains a jump to the `found` label. Note that as the zoom level increases, the widths of the proof steps tend to decrease, because the individual steps are accessing fewer resources.

We note one novel convention adopted in this proof: whenever a ribbon in a command's postcondition also appears in its precondition, and the size and positioning makes the correspondence unambiguous, the label in the postcondition can be replaced by a 'ditto' mark.

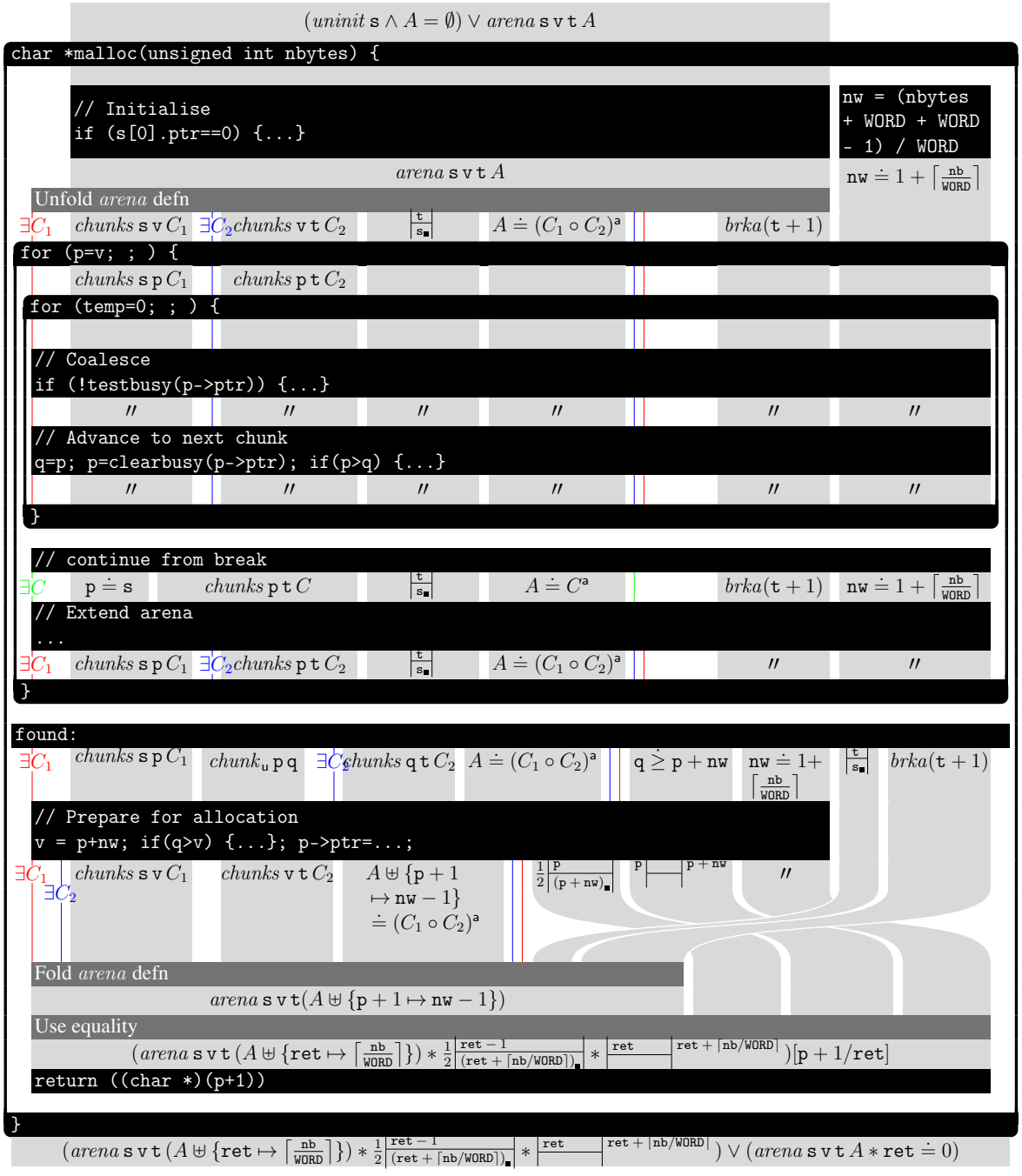


Figure 5.17: Ribbon proof of `malloc`, low detail. This diagram provides an overview proof of the entire routine.

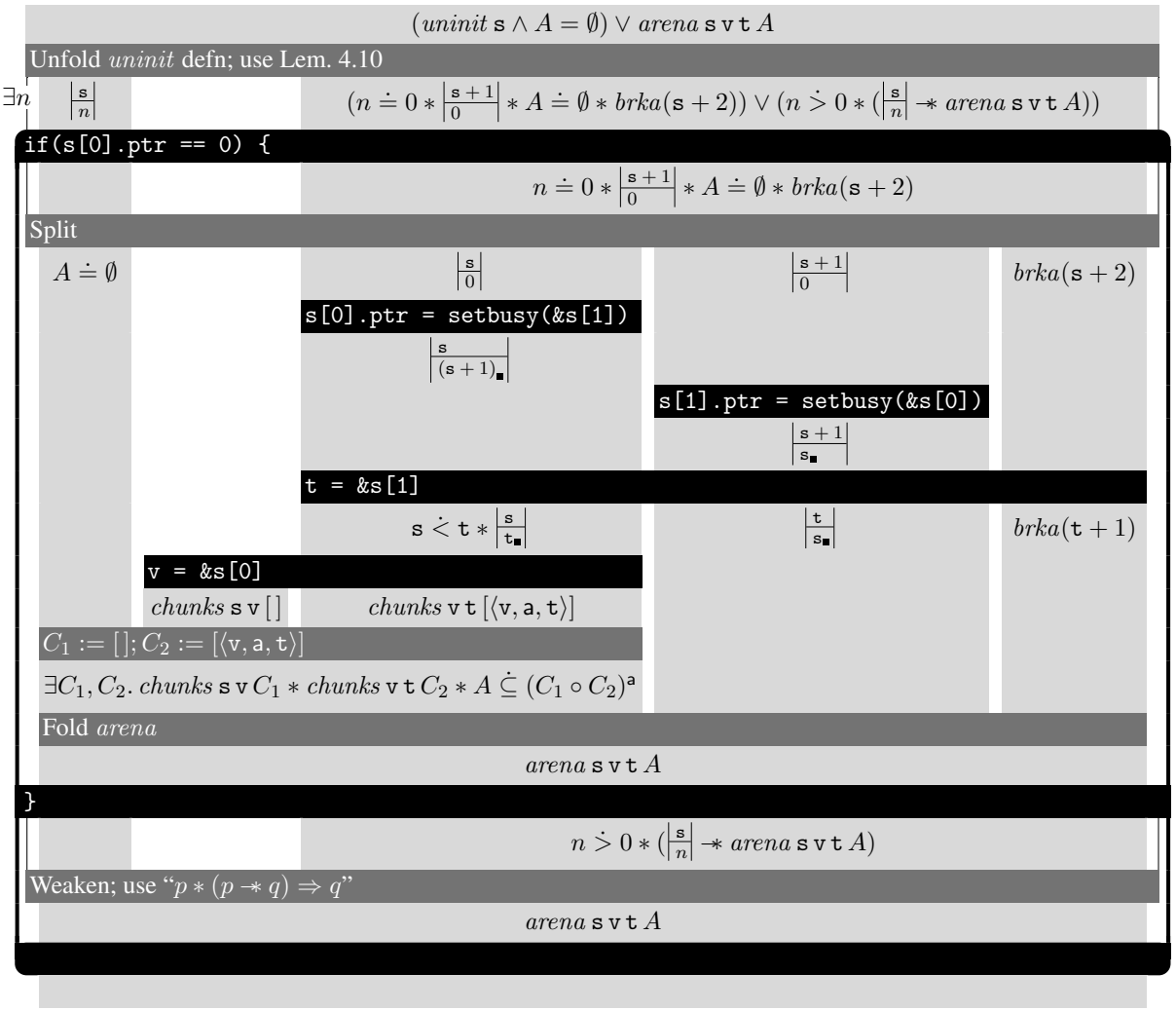


Figure 5.18: Ribbon proof of `malloc`, medium detail. This proof expands the 'Initialise' step in Fig. 5.17.

Figure 5.19: Ribbon proof of `malloc`, medium detail. This proof expands the 'Coalesce' step in Fig. 5.17.

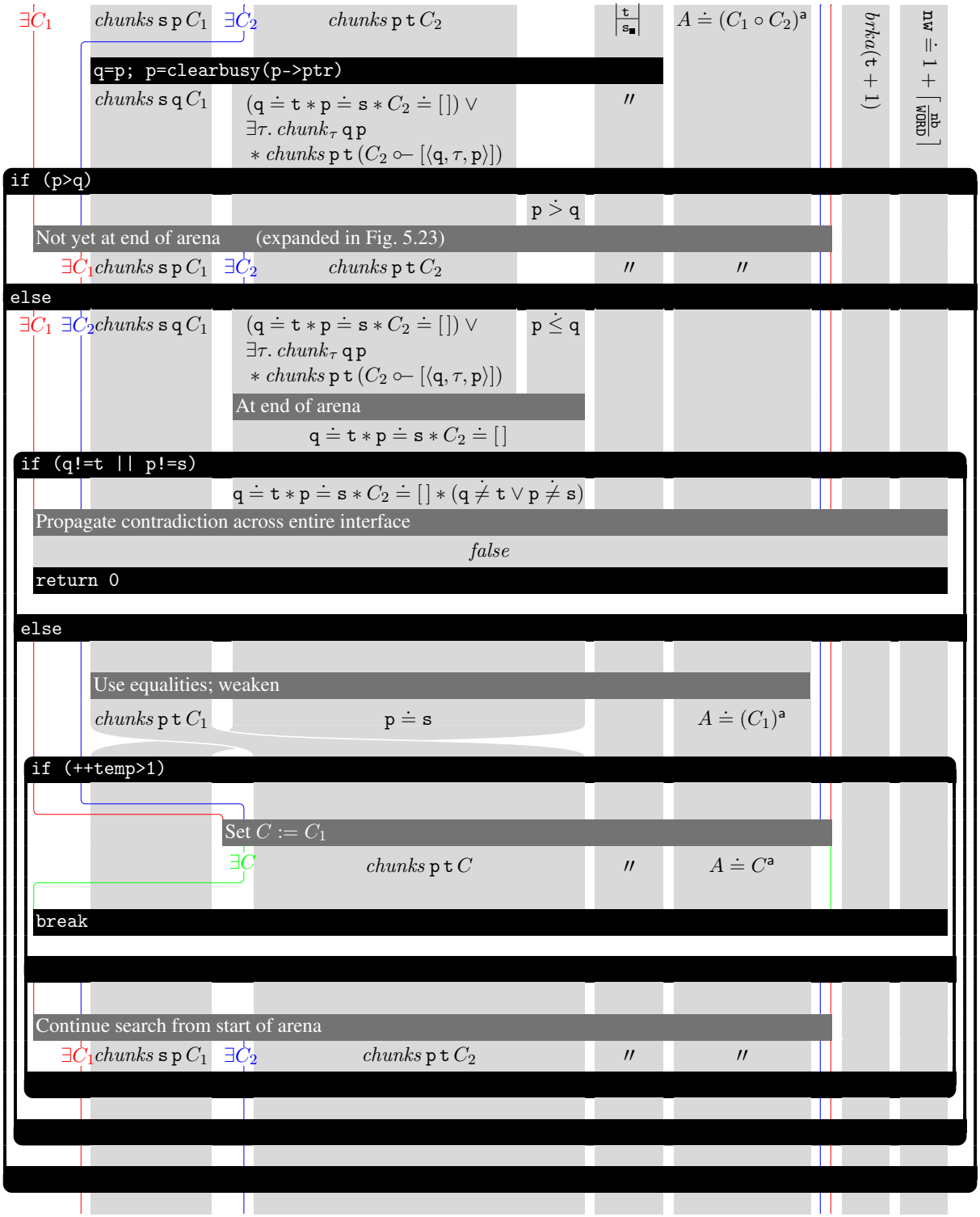


Figure 5.20: Ribbon proof of `malloc`, medium detail. This proof expands the 'Advance to next chunk' step in Fig. 5.17.

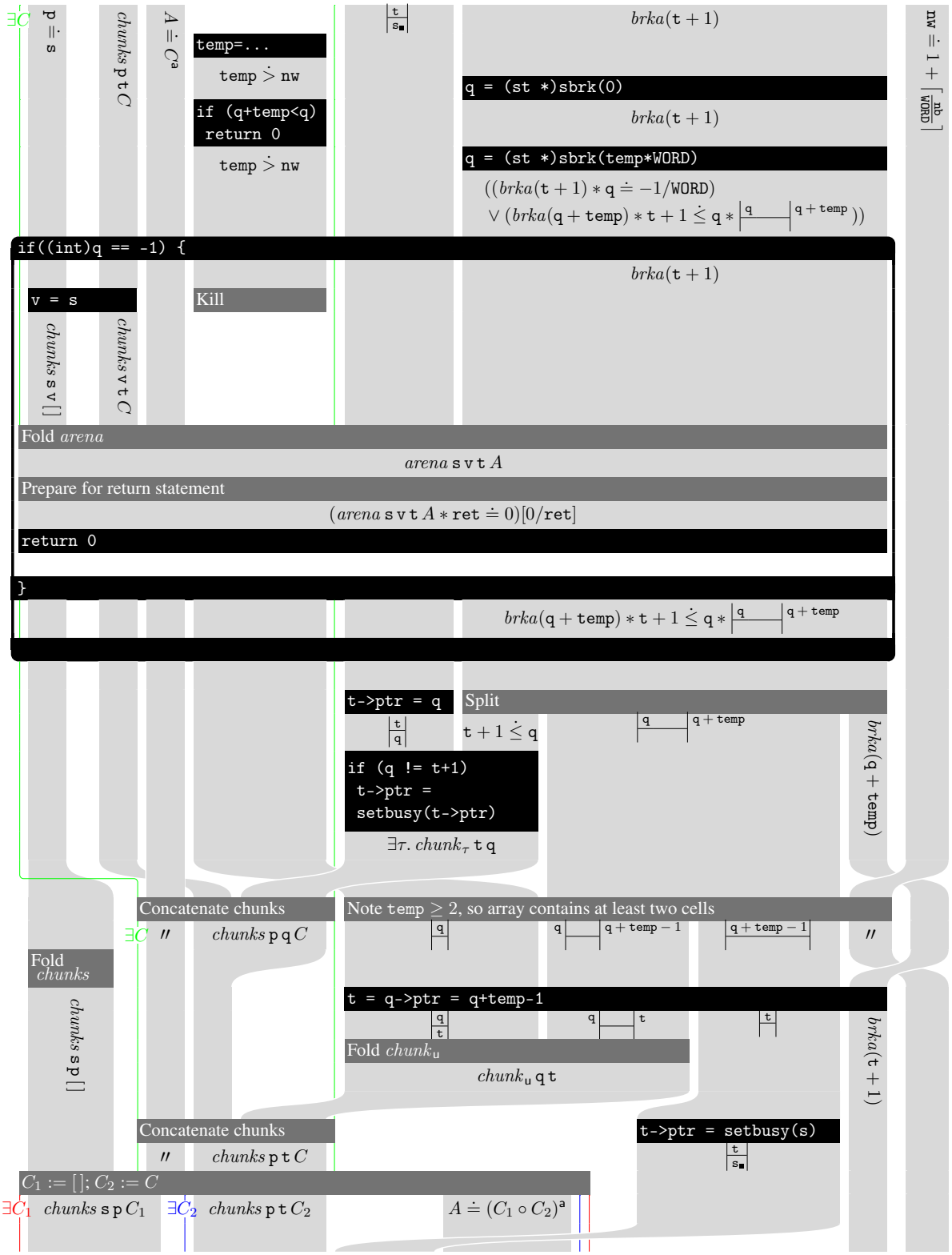


Figure 5.21: Ribbon proof of `malloc`, medium detail. This proof expands the 'Extend arena' step in Fig. 5.17.

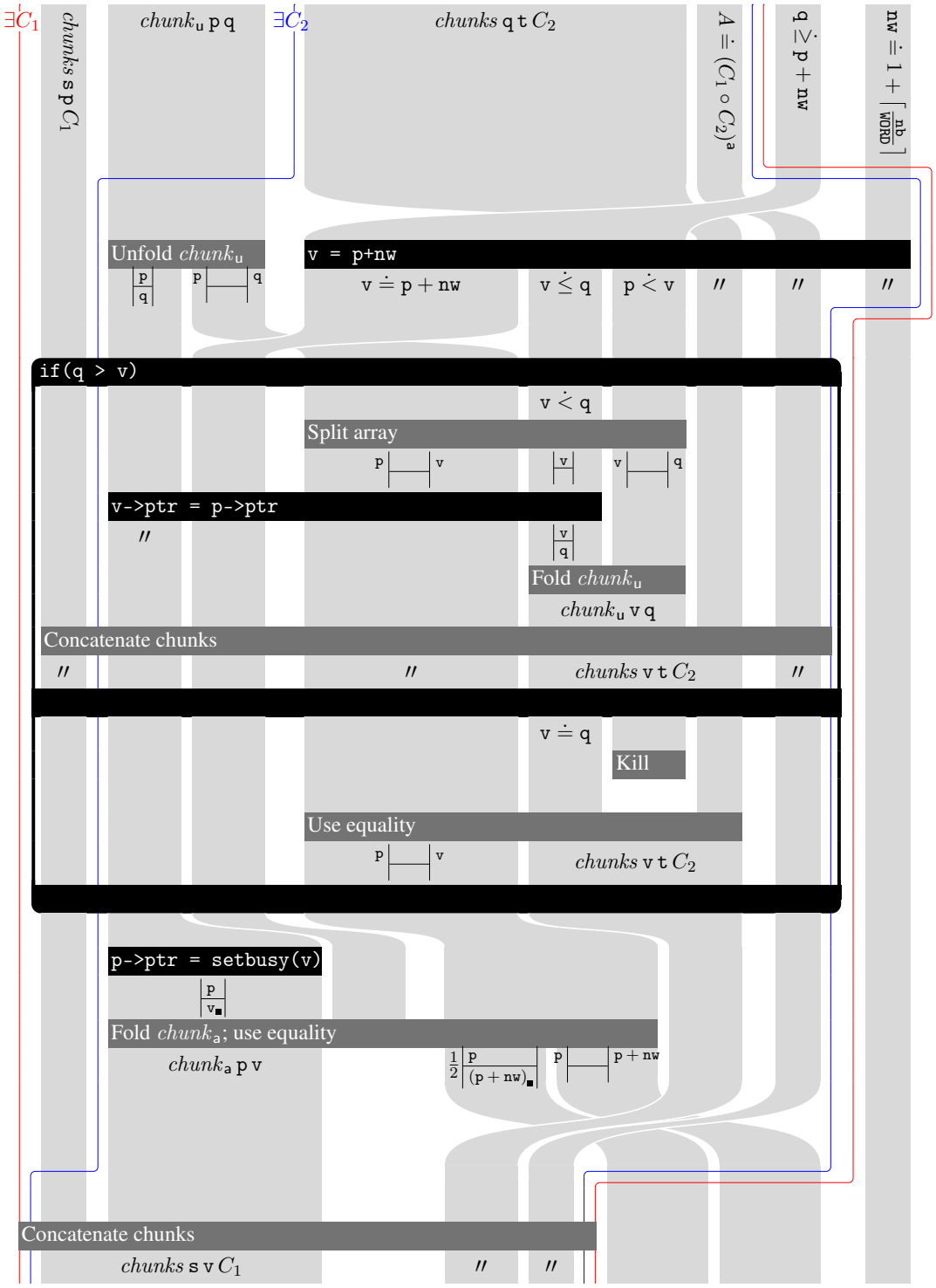


Figure 5.22: Ribbon proof of `malloc`, medium detail. This proof expands the 'Prepare for allocation' step in Fig. 5.17.

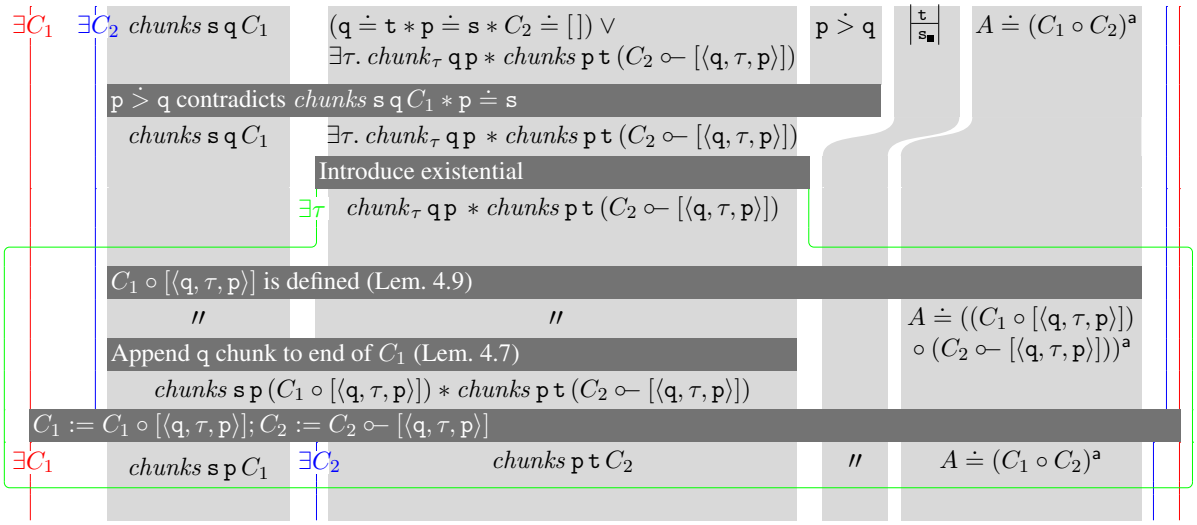


Figure 5.23: Ribbon proof of `malloc`, high detail. This proof expands the 'Not yet at end of arena' step in Fig. 5.20.

5.6 Tool support

Several properties of ribbon proofs make them potentially appealing as a partner for automatic verification tools based on separation logic, such as Bedrock [Chlipala 2011] and Verifast [Jacobs et al. 2011a]. Because ribbon proofs can be decomposed both horizontally and vertically, into independent proof blocks, they may suggest more opportunities for modular verification. One problem with automation is that users can lose track of their position in the proof: ribbons could provide an interface to the proof as it develops. Moreover, when automation fails, partial ribbon proofs could be used to view and guide the process manually. Ribbon proofs also shift the bureaucracy of rearranging assertions (in accordance with the associativity and commutativity of the $*$-operator) from the individual proof steps into the surrounding graphical structure, where it is more naturally handled.

To demonstrate the potential of ribbon proofs to complement automation, we have developed a prototype tool whose inputs are a ribbon diagram and a collection of small Isabelle proof scripts, one for each basic step. Our tool uses our Isabelle formalisation of Thm. 5.6 and the proof rules of Fig. 5.9 to assemble the Isabelle proof scripts for the individual commands into a single script that verifies the entire diagram.

Supplied with appropriate proof rules for primitive commands and a collection of axioms about lists, our tool has successfully verified the ribbon proofs in Figs. 5.3 and 5.5b. In both cases, all of the proof scripts for the individual basic steps are small, and they can often be discharged without manual assistance. Individual proof scripts can be checked in any order – even concurrently. This feature recalls recent developments in theorem proving that allow proofs to be processed in a non-serial manner [Wenzel 2012].

The input to the tool is a graphical ribbon diagram, following Defn. 5.8. Our tool begins by converting this graphical diagram into a stratified diagram, resolving any ambiguity about the node order by reference to the order of their input. By taking this approach, we avoid having to invest in variables-as-resource.

Our tool outputs a pictorial representation of the graph it has verified, laid out using the *dot*

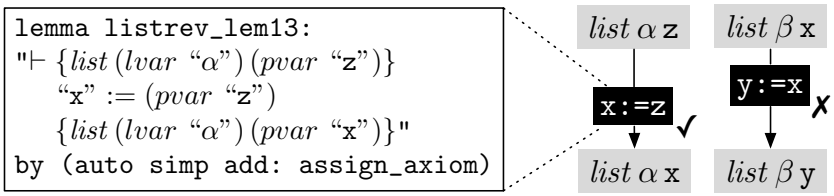


Figure 5.24: Tool support for checking ribbon proofs

tool in the *Graphviz* library.[1] One such picture (with the layout manually tweaked) is shown in Fig. 5.12. Clicking on any basic step loads the corresponding Isabelle proof script, which can then be edited. When a step's proof is admitted by Isabelle, the corresponding node in the pictorial representation is marked with a tick; a failed or incomplete proof is marked with a cross. Figure 5.24 illustrates this with an incomplete proof of a snippet of Fig. 5.12, and shows the Isabelle script for one of the steps.

In the current prototype, the user must supply the input in textual form, but in the future, we intend to enable direct interaction with the graphical representation, perhaps through a framework for diagrammatic reasoning such as Diabelli [Urbas and Jamnik 2012]. We envisage an interactive graphical interface for exploring and modifying proofs, that allows steps to be collapsed or expanded to the desired granularity: whether that is the fine details of every rule and axiom, or a coarse bird's-eye view of the overall structure of the proof.

The ribbon proofs in this chapter have all been laid out manually (and we are preparing a public release of the LaTeX macros we use to do this) but there is scope for additional tool support for discovering pleasing layouts automatically.

5.7 Related and further work

Ribbon proofs are more than just a pretty syntax; they are a sound and complete proof system. Proof outlines have previously been promoted from a notational device to a formal system by Schneider [1997], and by Ashcroft [1976], who remarks that "the essential property of [proof outlines] is that each piece of program appears *once*." Very roughly speaking, ribbon proofs extend this property to each piece of assertion.

When constructing a proof outline, one can reduce the repetition by 'framing off' state that is unused for several instructions. For instance, Fig. 5.25a depicts one variation of Fig. 5.1a obtained by framing off `x` during the latter two instructions; another option is to frame off `z` during the first two (Fig. 5.25b). It is unsatisfactory that there are several different proof outlines for what is essentially the same proof. More pragmatically, deciding among these options can be difficult with large proof outlines. Happily, each of these options yields the same ribbon proof (Fig. 5.1b). We note a parallel here with *proof nets* [Girard 1987], which are a graphical mechanism for unifying proofs in linear logic that differ only in uninteresting ways, such as the order of rule applications.

The graphical structures described in Defn. 5.8 resemble Milner's *bigraphs* [Milner 2009], with assertions and commands as nodes, a link graph to show the deductions of the proof, and a place graph to allow existential boxes, choices and loops to contain nested graphs. In fact, our diagrams correspond to a restricted form called *binding* bigraphs, in which edges may not cross place boundaries. Relaxing this restriction may enable a model of the 'dynamic' scoping

[1] `http://www.graphviz.org`

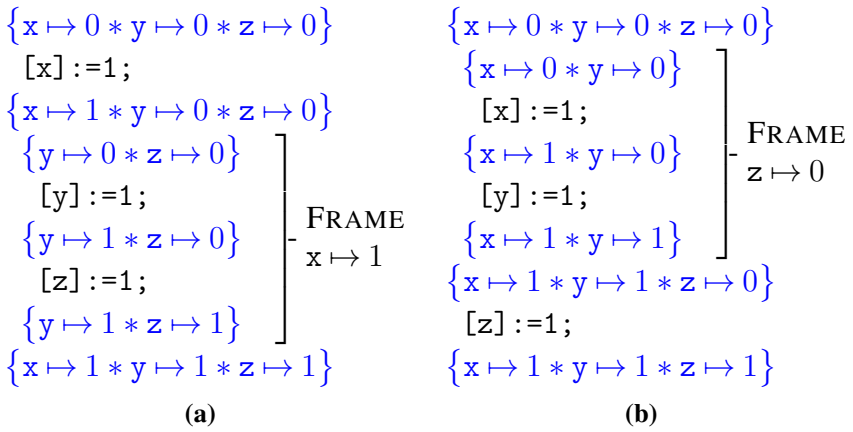


Figure 5.25: Two alternatives to the proof outline in Fig. 5.1a

of existential boxes exhibited in Fig. 5.6, which our current formalisation dismisses as a purely syntactic artefact.

Ribbon proofs can be understood as objects of a symmetric monoidal category, and our pictures as *string diagrams*. String diagrams are widely used as graphical languages for such categories [Selinger 2011]. In future work we intend to investigate this categorical semantics of ribbon proofs; in particular, the use of *traces* [Joyal et al. 1996] to model the loop construction depicted in Fig. 5.4b, and coproducts to model if-statements and existential boxes.

Another avenue for future work is to investigate the connection between our ribbon proofs and the *labelled separation logic* due to Raza et al. [2009]. Labelled separation logic seeks to justify compiler reorderings by analysing the dependencies between program instructions, and checking that these are not violated. The dependencies are detected by first labelling each component of each assertion with the instructions that access it, and then propagating these labels through program proofs. Raza et al.'s labels play a similar role to the *columns* in our ribbon diagrams: each ribbon and each instruction occupies one or more columns of a diagram, and instructions that occupy common columns may share a dependency (modulo ribbon twisting, which upsets the column ordering).

We have so far considered only sequential programs, even though the proofs themselves have a concurrent nature. It may be possible to extend our ribbon proof system to handle *concurrent separation logic* [O'Hearn 2004; Brookes 2004] as follows. Recall the single-cell buffer program considered on page 38. Figure 5.26 imagines a ribbon proof of the consumer thread of that program. The resource invariant is initially placed in a protected ribbon that is inaccessible to the thread (as suggested by the diagonal hatching). Upon entering the critical region, the ribbon becomes available, and upon leaving it, the resource invariant is re-established and the ribbon becomes inaccessible once again.

Beyond concurrent separation logic, we intend our proof system to be applied fruitfully to more advanced separation logics. It has already been applied to a logic for relaxed memory [Bornat and Dodds 2012]; some other candidates handle fine-grained concurrency [Feng et al. 2007; Vafeiadis and Parkinson 2007; Feng 2009; Dinsdale-Young et al. 2010], dynamic threads [Dodds et al. 2009], storable locks [Gotsman et al. 2007], loadable modules [Jacobs et al. 2011b] and garbage collection [Hur et al. 2011]. Increasingly complicated logics for increasingly complicated programming features make techniques for intuitive construction and clear presentation ever more crucial.

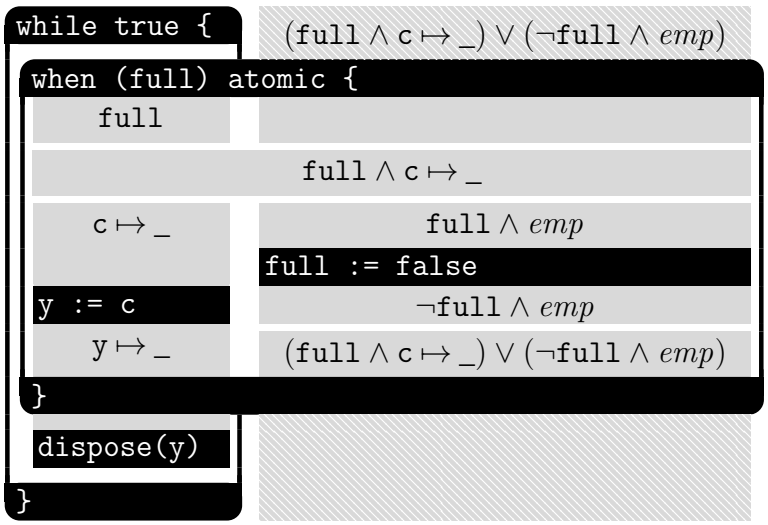


Figure 5.26: Ribbon proof of single-cell buffer (consumer thread)

Chapter 6

Outlook

This dissertation has made three main contributions.

Chapter 3 introduced explicit stabilisation, a re-imagining of the important concept of stability in rely-guarantee reasoning. First, we showed (in Sect. 3.2) how it can be used to simplify complex rely-guarantee proof rules. Although the particular proof rule on which this simplification was demonstrated is useful only in a rather limited domain, we believe that explicit stabilisation is also applicable to a range of more general program logics based on rely-guarantee. Second, we showed how explicit stabilisation underpins a new version of rely-guarantee that permits, for the first time, reasoning about library code. A potential downside of this so-called 'parametric rely-guarantee' system is that assertions become more complex: they are functions, parameterised by a rely relation. This complexity should be surmountable however, since program logics that treat assertions as relations have previously achieved acceptance. Jones [1990], for instance, has $p \wedge q$ really standing for $\lambda(\sigma, \sigma').\, p(\sigma') \wedge q(\sigma')$. Third, we showed how to encode three variants of rely-guarantee reasoning – early, mid and late stability – into parametric rely-guarantee. By unifying these previously-disparate variants, we provide insights about the essence of rely-guarantee reasoning, and further our case that explicit stabilisation is a natural and useful way to think about stability.

Chapter 4 identified the difficulties with verifying a sequential module – such as the Version 7 Unix memory manager – that exposes to its clients some information about its internal state. We showed how to adapt the RGSep logic to the task of verifying such modules, and how the addition of explicit stabilisation allowed the module's 'internal interference' to be hidden while verifying its clients. We find this combination of RGSep and explicit stabilisation to be an interesting study into the verification of sequential modules, but find that several technical details – principally, the restriction to globally-scoped modules – render the technique flawed. We intend in future work to investigate replacing RGSep with *concurrent abstract predicates*, which should lift this restriction.

Chapter 5 introduced ribbon diagrams as an attractive and practical approach for visualising program proofs conducted in separation logic or any derivative thereof. We find them to be a more readable, informative and flexible representation of a proof than the traditional 'proof outline'. Each step of the proof can be checked locally, by focusing only on the relevant resources. Ribbon proofs show graphically the distribution of resource in a program, and in particular, which parts of a program operate on disjoint resources, and this may prove useful for exploring parallelisation opportunities. Because they are a much less redundant representation, they also have the potential for greater scalability. We demonstrated this with a ribbon proof of a fairly complex program – the Version 7 Unix memory manager – but we concede that more work remains to make such large ribbon proofs truly readable. To this end, we intend in future work

to build tool support for viewing and manipulating hierarchically-structured ribbon proofs.

Appendix A

Supplementary material

A.1 Proof of Theorem 5.5

Theorem 5.5. *The following rules are derivable from those in Fig. 5.9.*

$$\textsc{SeqComp}\quad \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to P' \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to P''}{\vdash^{\text{rdia}}_{\mathsf{SL}} \begin{matrix} D \\ E \end{matrix} : P \to P''} \qquad \textsc{ParComp}\quad \frac{\vdash^{\text{rdia}}_{\mathsf{SL}} D : P \to Q \qquad \vdash^{\text{rdia}}_{\mathsf{SL}} E : P' \to Q' \qquad D \# E}{\vdash^{\text{rdia}}_{\mathsf{SL}} D \parallel_\mu E : P\,P' \to Q\,Q'}$$

Proof of SeqComp *rule.* Suppose $D = [D_0, \ldots, D_k]$ and $E = [E_0, \ldots, E_l]$ for non-negative k and l. By rule inversion on RMain, we obtain:

$$\forall i \in k+1.\, \vdash^{\text{cell}}_{\mathsf{SL}} D_i : Q_i \to Q_{i+1} \tag{A.1}$$

$$\forall i \in l+1.\, \vdash^{\text{cell}}_{\mathsf{SL}} E_i : R_i \to R_{i+1} \tag{A.2}$$

for some $[Q_0, \ldots, Q_{k+1}]$ and $[R_0, \ldots, R_{l+1}]$ with $Q_0 = P$, $Q_{k+1} = P' = R_0$ and $R_{l+1} = P''$. Now define a list

$$[S_0, \ldots, S_{k+l+1}]$$

such that:

$$S_i = \begin{cases} Q_i & \text{if } 0 \le i \le k+1 \\ R_{i-k-1} & \text{if } k+1 \le i \le l+1, \end{cases}$$

noting that $S_{k+1} = Q_{k+1} = R_0$. By the RMain rule, it suffices to show:

$$\forall i \in k+l+2.\, \vdash^{\text{cell}}_{\mathsf{SL}} \left(\begin{smallmatrix} D \\ E \end{smallmatrix}\right)_i : S_i \to S_{i+1}.$$

If $i < k+1$, then $\left(\begin{smallmatrix} D \\ E \end{smallmatrix}\right)_i = D_i$, $S_i = Q_i$ and $S_{i+1} = Q_{i+1}$, so the result follows from (A.1). Otherwise, if $k+1 \le i < k+l+2$, then $\left(\begin{smallmatrix} D \\ E \end{smallmatrix}\right)_i = E_{i-k-1}$, $S_i = R_{i-k-1}$ and $S_{i+1} = R_{i-k}$, so the result follows from (A.2). □

For proving the ParComp rule, we shall require a little more machinery. We employ the following generalisation of the Hoare triple.

Definition A.1 (Hoare chain)**.** A *Hoare chain* Π is a term of the following language:

$$\Pi ::= \{P\} \mid \{P\}\,(\gamma, F)\,\Pi$$

where $P, F \in \mathsf{Interface}$ and $\gamma \in \mathsf{Cell}$. A Hoare chain of length k can be written

$$\{P_0\}\,(\gamma_0, F_0)\,\{P_1\}\,\cdots\,\{P_{k-1}\}\,(\gamma_{k-1}, F_{k-1})\,\{P_k\}.$$

If this chain is called Π, then we define $pre(\Pi)$ as P_0 and $post(\Pi)$ as P_k.

Definition A.2 (Provability of a Hoare chain)**.** A chain is provable, written $\vdash_{\mathsf{SL}}^{\mathrm{chain}} \Pi$, if each of its triples is provable; that is:

$$\begin{array}{lcl} \vdash_{\mathsf{SL}}^{\mathrm{chain}} \{P\} & = & \mathit{true} \\ \vdash_{\mathsf{SL}}^{\mathrm{chain}} \{P\}\,(\gamma, F)\,\Pi & = & \left(\vdash_{\mathsf{SL}}^{\mathrm{rdia}} (\gamma, F) : P \to \mathit{pre}(\Pi)\right) \text{ and } \vdash_{\mathsf{SL}}^{\mathrm{chain}} \Pi. \end{array}$$

Definition A.3 (Extracting a Hoare chain from a rasterised diagram)**.** Note that the empty list is not a rasterised diagram.

$$\begin{array}{lcl} \mathit{chain}[(\gamma, F)] & = & \{\mathit{top}\,\gamma \otimes F\}\,(\gamma, F)\,\{\mathit{bot}\,\gamma \otimes F\} \\ \mathit{chain}((\gamma, F) :: D) & = & \{\mathit{top}\,\gamma \otimes F\}\,(\gamma, F)\,(\mathit{chain}\,D). \end{array}$$

Lemma A.4. *We have* $pre(\mathit{chain}\,D) = \mathit{top}\,D$ *and* $post(\mathit{chain}\,D) = \mathit{bot}\,D$.

Lemma A.5. *We have* $\vdash_{\mathsf{SL}}^{\mathrm{chain}} (\mathit{chain}\,D)$ *if and only if* $\vdash_{\mathsf{SL}}^{\mathrm{rdia}} D$.

Proof. By structural induction on D. □

Definition A.6 (Parallel composition of Hoare chains)**.** If Π_0 and Π_1 are Hoare chains of lengths k and l, and μ is a sequence containing k zeroes and l ones, then $\Pi_0 \parallel_\mu \Pi_1$ is defined according to the following equations:

$$\begin{array}{rcl} \{P\} \parallel_\epsilon \{Q\} & = & \{P \otimes Q\} \\ (\{P\}\,(\gamma, F)\,\Pi_0) \parallel_{0\mu} \Pi_1 & = & \{P \otimes \mathit{pre}(\Pi_1)\}\,(\gamma, F \otimes \mathit{pre}(\Pi_1))\,(\Pi_0 \parallel_\mu \Pi_1) \\ \Pi_0 \parallel_{1\mu} (\{Q\}\,(\gamma, F)\,\Pi_1) & = & \{\mathit{pre}(\Pi_0) \otimes Q\}\,(\gamma, \mathit{pre}(\Pi_0) \otimes F)\,(\Pi_0 \parallel_\mu \Pi_1). \end{array}$$

Lemma A.7. *For any* $k \geq 0$*, for any binary sequence* μ *containing* $k_0 + 1$ *zeroes and* $k_1 + 1$ *ones, where* $k = k_0 + k_1$*, and for any provable rasterised diagrams* D *(of length* $k_0 + 1$*) and* E *(of length* $k_1 + 1$*):*

$$\begin{array}{rcl} \mathit{chain}(D \parallel_\mu E) & = & \mathit{chain}(D) \parallel_\mu \mathit{chain}(E) \\ \mathit{top}(D \parallel_\mu E) & = & \mathit{top}(D) \otimes \mathit{top}(E) \\ \mathit{bot}(D \parallel_\mu E) & = & \mathit{bot}(D) \otimes \mathit{bot}(E). \end{array}$$

Proof. By mathematical induction on k. In the base case, μ is either 01 or 10. In the inductive step, μ is either $0\mu'$ or $1\mu'$, for some μ' containing at least one zero and at least one one. □

Lemma A.8. *If* $\vdash^{\text{cell}}_{\mathsf{SL}}(\gamma, F) : P \to Q$ *and* $wr(\gamma) \not\frown rd(R)$ *then* $\vdash^{\text{cell}}_{\mathsf{SL}}(\gamma, F \otimes R) : P \otimes R \to Q \otimes R$.

Proof. By rule induction on $\vdash^{\text{cell}}_{\mathsf{SL}}$. □

Lemma A.9. *For any* $k \geq 0$*, for any binary sequence* μ *containing* k_0 *zeroes and* k_1 *ones, where* $k = k_0 + k_1$*, and for any chains* Π_0 *and* Π_1 *of lengths* k_0 *and* k_1*, if* $\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_0$ *and* $\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_1$ *and* $\Pi_0 \# \Pi_1$ *then* $\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_0 \parallel_\mu \Pi_1$ *and* $pre(\Pi_0 \parallel_\mu \Pi_1) = pre(\Pi_0) \otimes pre(\Pi_1)$.

Proof. By mathematical induction on k. When $k = 0$, then $k_0 = k_1 = 0$, so Π_0 and Π_1 both comprise single interfaces, say $\{P\}$ and $\{Q\}$. Hence $\Pi_0 \parallel_\mu \Pi_1 = \{P \otimes Q\}$, which is vacuously provable. For the inductive step, assume $k = 1 + k'$ for some $k' \geq 0$. Then μ is non-empty, and hence begins with 0 or 1. Suppose it begins with 0; the alternative case is argued similarly. That is, $\mu = 0\mu'$ for some μ'. We deduce $k_0 > 0$, which means Π_0 can be written as

$$\{P\}\,(\gamma, F)\,\Pi_0'.$$

for some P, γ, F and Π_0'. Since Π_0 is provable, then so is Π_0', and

$$\vdash^{\text{cell}}_{\mathsf{SL}}(\gamma, F) : P \to pre(\Pi_0') \tag{A.3}$$

holds. Now, $\Pi_0 \parallel_\mu \Pi_1$ is equal to:

$$\{P \otimes pre(\Pi_1)\}\,(\gamma, F \otimes pre(\Pi_1))\,(\Pi_0' \parallel_{\mu'} \Pi_1)$$

by Defn. A.6. This Hoare chain is provable if

$$\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_0' \parallel_{\mu'} \Pi_1 \tag{A.4}$$

$$\vdash^{\text{cell}}_{\mathsf{SL}}(\gamma, F \otimes pre(\Pi_1)) : P \otimes pre(\Pi_1) \to pre(\Pi_0' \parallel_{\mu'} \Pi_1). \tag{A.5}$$

But (A.4) holds as a direct result of the induction hypothesis. The induction hypothesis also allows (A.5) to be written as:

$$\vdash^{\text{cell}}_{\mathsf{SL}}(\gamma, F \otimes pre(\Pi_1)) : P \otimes pre(\Pi_1) \to pre(\Pi_0') \otimes pre(\Pi_1)$$

which follows from (A.3) via Lem. A.8, noting that the side-condition on variable interference is met having assumed $\Pi_0 \# \Pi_1$. □

Proof of PARCOMP *rule.* The soundness of the following rule:

$$\frac{\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_0 \qquad \vdash^{\text{chain}}_{\mathsf{SL}} \Pi_1 \qquad \Pi_0 \# \Pi_1}{\vdash^{\text{chain}}_{\mathsf{SL}} \Pi_0 \parallel_\mu \Pi_1}$$

follows from Lem. A.9. The PARCOMP rule can be derived from this rule, together with Lems. A.5 and A.7. □

A.2 Proof of Theorem 5.13

Theorem 5.13. *The following rules are derivable from those in Fig. 5.13.*

GSEQCOMP

$$\frac{\vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q \quad \vdash^{\text{dia}}_{\mathsf{SL}} H : Q \to R}{\vdash^{\text{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix} : P \to R}$$

GPARCOMP

$$\frac{\vdash^{\text{dia}}_{\mathsf{SL}} G : P \to Q \quad \vdash^{\text{dia}}_{\mathsf{SL}} H : P' \to Q'}{\vdash^{\text{dia}}_{\mathsf{SL}} G \parallel H : P\,P' \to Q\,Q'}$$

When sequentially composing diagrams, we shall need to rename nodes.

Definition A.10 (Support equivalence)**.** Two diagrams G and H are support-equivalent, written $G \simeq H$, iff there exists a bijection $\rho : V_G \to V_H$ that satisfies $\Lambda_G = \Lambda_H \circ \rho$, and for all $\mathbf{v}, \chi, \mathbf{w}$:

$$(\mathbf{v}, \chi, \mathbf{w}) \in E_G \quad \Leftrightarrow \quad (\{\rho\, v \mid v \in \mathbf{v}\}, \chi, \{\rho\, w \mid w \in \mathbf{w}\}) \in E_H.$$

Lemma A.11. *For any diagrams G and H, if $G \simeq H$ then $top(G) = top(H)$ and $bot(G) = bot(H)$.*

Lemma A.12. *For any diagrams G and H, if $G \simeq H$ then $\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q = \vdash^{\mathrm{dia}}_{\mathsf{SL}} H : P \to Q$.*

Proof. Suppose $\vdash^{\mathrm{dia}}_{\mathsf{SL}} G : P \to Q$. Perform rule-inversion on MAIN, apply the properties given in Defn. A.10 and Lem. A.11, then re-apply MAIN. □

Building on Defns. 5.11 and 5.12, we now provide more careful definitions of sequential and parallel composition that take node-renaming into account. (Technically, our definitions do not describe functions, because the operations can result in several different diagrams. However, since these diagrams are all support-equivalent, it is reasonable to think of sequential and parallel composition as functions.)

Definition A.13 (Sequential composition of diagrams – amended)**.** We notate sequential composition by vertical stacking. We overload this notation for both diagrams and assertion-gadgets. If G and H are diagrams, and there exists $H' \simeq H$ such that:

- $\mathit{terminals}\, G = \mathit{initials}\, H' = V_G \cap V_{H'}$, and
- $\begin{matrix}\Lambda_G(v)\\ \Lambda_{H'}(v)\end{matrix}$ is defined for all $v \in V_G \cap V_{H'}$

then we write $\begin{matrix}G\\ H\end{matrix}$ for the diagram $(V_G \cup V_{H'}, \Lambda, E_G \cup E_{H'})$, where

$$\Lambda(v) = \begin{cases} \Lambda_G(v) & \text{if } v \in V_G \setminus V_{H'} \\ \Lambda_{H'}(v) & \text{if } v \in V_{H'} \setminus V_G \\ \begin{pmatrix}\Lambda_G(v)\\ \Lambda_{H'}(v)\end{pmatrix} & \text{if } v \in V_G \cap V_{H'}. \end{cases}$$

Definition A.14 (Parallel composition of diagrams – amended)**.** If G and H are diagrams, and there exists $H' \simeq H$ such that $V_G \not\frown V_{H'}$, then we write $G \parallel H$ for the diagram

$$(V_G \cup V_{H'}, \Lambda_G \cup \Lambda_{H'}, E_G \cup E_{H'}).$$

Our strategy for proving the soundness of the GSEQCOMP rule is induction on the structure of the first component, G. This structure is mutually recursive, comprising graphical diagrams and command-gadgets and assertion-gadgets. Hence, we must prove the following stronger lemma, which makes one statement for each type of structure. Only the first of these, Φ_{dia}, is required for demonstrating the soundness of the GSEQCOMP rule.

Lemma A.15. *Let:*

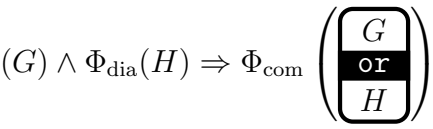
$$\begin{aligned}
\Phi_{\mathrm{dia}}(G) &\stackrel{\mathrm{def}}{=} \forall H.\, \textit{if } bot(G) = top(H) \textit{ then} \\
&\qquad (\mathrm{defined}\,\tbinom{G}{H} \textit{ and } (\textit{if } \vdash^{\mathrm{dia}}_{\mathsf{SL}} G \textit{ and } \vdash^{\mathrm{dia}}_{\mathsf{SL}} H \textit{ then} \\
&\qquad\quad \vdash^{\mathrm{dia}}_{\mathsf{SL}} \tbinom{G}{H} \textit{ and } bot(\tbinom{G}{H}) = bot(H) \textit{ and } top(\tbinom{G}{H}) = top(G))) \\
\Phi_{\mathrm{asn}}(A) &\stackrel{\mathrm{def}}{=} \forall B.\, \textit{if } bot(A) = top(B) \textit{ then} \\
&\qquad (\mathrm{defined}\,\tbinom{A}{B} \textit{ and } (\textit{if } \vdash^{\mathrm{asn}}_{\mathsf{SL}} A \textit{ and } \vdash^{\mathrm{asn}}_{\mathsf{SL}} B \textit{ then} \\
&\qquad\quad \vdash^{\mathrm{asn}}_{\mathsf{SL}} \tbinom{A}{B} \textit{ and } bot(\tbinom{A}{B}) = bot(B) \textit{ and } top(\tbinom{A}{B}) = top(A))) \\
\Phi_{\mathrm{com}}(\chi) &\stackrel{\mathrm{def}}{=} \textit{true}.
\end{aligned}$$

Then we have

$$\Phi_{\mathrm{dia}}(G) \wedge \Phi_{\mathrm{asn}}(A) \wedge \Phi_{\mathrm{com}}(\chi)$$

for all diagrams G, assertion-gadgets A and command-gadgets χ.

Proof. We proceed by structural induction on diagrams. The six cases are as follows.

1. $\forall p.\, \Phi_{\mathrm{asn}}($ $)$
2. $\forall x, G.\, \Phi_{\mathrm{dia}}(G) \Rightarrow \Phi_{\mathrm{asn}}($  $)$
3. $\forall c.\, \Phi_{\mathrm{com}}($  $)$
4. $\forall G, H.\, \Phi_{\mathrm{dia}}(G) \wedge \Phi_{\mathrm{dia}}(H) \Rightarrow \Phi_{\mathrm{com}}$ 
5. $\forall G.\, \Phi_{\mathrm{dia}}(G) \Rightarrow \Phi_{\mathrm{com}}$ 
6. $\forall G.\, (\forall v \in V_G.\, \Phi_{\mathrm{asn}}(\Lambda_G\, v)) \wedge (\forall (\_, \chi, \_) \in E_G.\, \Phi_{\mathrm{com}}(\chi)) \;\Rightarrow \Phi_{\mathrm{dia}}(G)$

Only the sixth is interesting. To show $\Phi_{\mathrm{dia}}(G)$, we start by picking an arbitrary H and assuming $bot(G) = top(H)$. That is,

$$\otimes_{v \in terminals(G)}\, bot(\Lambda_G\, v) = \otimes_{v \in initials(H)}\, top(\Lambda_H\, v).$$

Hence there exists a bijection $\pi : terminals(G) \to initials(H)$ for which:

$$\forall v \in terminals(G).\, bot(\Lambda_G\, v) = top(\Lambda_H(\pi\, v)).$$

We can apply the first of our two inductive hypotheses to this to obtain:

$$\forall v \in terminals(G).\, \mathrm{defined}\begin{pmatrix} \Lambda_G\, v \\ \Lambda_H(\pi\, v) \end{pmatrix}. \tag{A.6}$$

Now we pick a new diagram $H' \doteq H$, obtained by applying a node-renaming ρ to H that satisfies:

$$\begin{gathered}
\forall v \in initials(H).\, \rho(v) = \pi^{-1}(v) \\
\forall v \in V_H \setminus initials(H).\, \rho(v) \notin V_G
\end{gathered}$$

That is, ρ ensures that the initial nodes of H' coincide with the terminal nodes of G, and that its other nodes are disjoint from G's. We now have:

$$terminals(G) = initials(H') = V_G \cap V_{H'}.$$

With (A.6), we obtain:

$$\forall v \in V_G \cap V_{H'}.\, \text{defined} \begin{pmatrix} \Lambda_G\, v \\ \Lambda_{H'}\, v \end{pmatrix}.$$

These two facts are sufficient for establishing $\text{defined} \binom{G}{H}$. For the second part of $\Phi_{\text{dia}}(G)$, we must show

$$\vdash^{\text{dia}}_{\mathsf{SL}} \begin{matrix} G \\ H \end{matrix}$$

under the additional assumptions that $\vdash^{\text{dia}}_{\mathsf{SL}} G$ and $\vdash^{\text{dia}}_{\mathsf{SL}} H$ both hold. We use rule inversion on MAIN, and then Lem. A.12 to deduce:

$$\forall v \in V_G.\, \vdash^{\text{asn}}_{\mathsf{SL}} \Lambda_G\, v \tag{A.7}$$

$$\forall v \in V_{H'}.\, \vdash^{\text{asn}}_{\mathsf{SL}} \Lambda_{H'}\, v \tag{A.8}$$

$$\forall (\mathbf{v}, \chi, \mathbf{w}) \in E_G.\, \vdash^{\text{com}}_{\mathsf{SL}} \chi : \otimes_{v \in \mathbf{v}}\, bot(\Lambda_G\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda_G\, w) \tag{A.9}$$

$$\forall (\mathbf{v}, \chi, \mathbf{w}) \in E_{H'}.\, \vdash^{\text{com}}_{\mathsf{SL}} \chi : \otimes_{v \in \mathbf{v}}\, bot(\Lambda_{H'}\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda_{H'}\, w). \tag{A.10}$$

We are to show:

$$\forall v \in V_G \cup V_{H'}.\, \vdash^{\text{asn}}_{\mathsf{SL}} \Lambda\, v \tag{A.11}$$

$$\forall (\mathbf{v}, \chi, \mathbf{w}) \in E_G \cup E_{H'}.\, \vdash^{\text{com}}_{\mathsf{SL}} \chi : \otimes_{v \in \mathbf{v}}\, bot(\Lambda\, v) \to \otimes_{w \in \mathbf{w}}\, top(\Lambda\, w) \tag{A.12}$$

where Λ is as defined in Defn. A.13. To show (A.11), fix an arbitrary v in $V_G \cup V_{H'}$. If $v \in V_G \setminus V_{H'}$, use (A.7). If $v \in V_{H'} \setminus V_G$, use (A.8). For the case when $v \in V_G \cap V_{H'}$, we require

$$\vdash^{\text{asn}}_{\mathsf{SL}} \begin{pmatrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{pmatrix},$$

which is obtained from the inductive hypothesis. To show (A.12), fix an arbitrary edge $(\mathbf{v}, \chi, \mathbf{w})$ in $E_G \cup E_{H'}$. Suppose it is in E_G; the other possibility is handled similarly. We can use (A.9), but only once we have established

$$\otimes_{v \in \mathbf{v}}\, bot(\Lambda_G\, v) = \otimes_{v \in \mathbf{v}}\, bot(\Lambda\, v) \tag{A.13}$$

$$\otimes_{w \in \mathbf{w}}\, top(\Lambda_G\, w) = \otimes_{w \in \mathbf{w}}\, top(\Lambda\, w). \tag{A.14}$$

Of these, (A.13) follows from

$$\forall v \in \mathbf{v}.\, bot(\Lambda_G\, v) = bot(\Lambda\, v),$$

which holds because if $v \in V_G \setminus V_{H'}$ then Λ_G and Λ coincide, and if $v \in V_G \cap V_{H'}$ then v must be a terminal node of G and hence cannot be an incoming node of the edge $(\mathbf{v}, \chi, \mathbf{w})$. We obtain (A.14) analogously.

The final part of $\Phi_{\text{dia}}(G)$ requires $top(\binom{G}{H}) = top(H)$ and $bot(\binom{G}{H}) = bot(H)$. We give details only for the latter. After unfolding the definition of bot , it suffices to exhibit a bijection $\pi : terminals(H) \to terminals\, \binom{G}{H}$ such that:

$$\forall v \in terminals\, H.\, bot(\Lambda(\pi\, v)) = bot(\Lambda_H\, v).$$

In fact ρ, restricted to the terminal nodes of H, is such a bijection. It then suffices to show:

$$\forall v \in terminals\, H'.\, bot(\Lambda\, v) = bot(\Lambda_{H'}\, v).$$

This, in turn, is proved by cases. When $v \in V_{H'} \setminus V_G$ then Λ and $\Lambda_{H'}$ coincide by definition. When $v \in V_{H'} \cap V_G$, then $bot(\Lambda\, v)$ is equal to

$$bot \begin{pmatrix} \Lambda_G(v) \\ \Lambda_{H'}(v) \end{pmatrix},$$

which is equal to $bot(\Lambda_{H'}\, v)$ by the induction hypothesis. □

Proof of Thm. 5.13. The GSEQCOMP rule is a straightforward consequence of Lem. A.15. For the GPARCOMP rule, the key step is to show that

$$top(G \parallel H) = top(G) \otimes top(H)$$

Suppose that the composition operation renames H to H'. We can show

$$initials(G \parallel H) = initials(G) \uplus initials(H')$$

and hence:

$$\begin{aligned} top(G \parallel H) &= \otimes_{v \in initials(G \parallel H)}(top(\Lambda_{G \parallel H}\, v)) \\ &= (\otimes_{v \in initials\, G}(top(\Lambda_{G \parallel H}\, v))) \otimes (\otimes_{v \in initials\, H'}(top(\Lambda_{G \parallel H}\, v))) \\ &= (\otimes_{v \in initials\, G}(top(\Lambda_G\, v))) \otimes (\otimes_{v \in initials\, H'}(top(\Lambda_{H'}\, v))) \\ &= top(G) \otimes top(H') \\ &= top(G) \otimes top(H) \qquad \text{(by Lem. A.11).} \end{aligned}$$

□

A.3 Proof of Theorem 5.15

This proof has been formalised in Isabelle, and the proof script can be viewed online at:

`http://www.cl.cam.ac.uk/~jpw48/ribbons.html`

Theorem 5.15 (Soundness – variables-as-resource)**.** *Separation logic with variables-as-resource can encode any ribbon diagram that is provable with variables-as-resource:*

$$\vdash^{\text{gra}}_{\mathsf{VaR}} G : P \to Q \implies \forall c \in coms\, G. \vdash_{\mathsf{VaR}} \{asn\, P\}\, c\, \{asn\, Q\}.$$

Notation. For sets X and Y, let $X \uplus Y$ be defined when $X \not\cap Y$ as $X \cup Y$, and $X - Y$ be defined when $Y \subseteq X$ as $X \setminus Y$.

To prove this theorem, we employ the following generalisation of a Hoare triple. Note that this definition is unrelated to that given in Defn. A.1.

Definition A.16 (Hoare chain)**.** A Hoare chain is a sequence

$$\{P_0\}\, x_0\, \{P_1\} \cdots \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

where each P_i is an Interface, and each x_i is either a ComGadget or an AsnGadget.

Definition A.17 (Extracting Hoare chains)**.** First, we define the notion of a *proof state*. At any point while stepping through a Hoare chain, a proof state $\sigma \subseteq V_G \times \{\text{TOP}, \text{BOT}\}$ records those node-identifiers which are either initial or have been produced as the postcondition of an already-processed hyperedge and not yet consumed as a precondition of another. A node-identifier is tagged BOT if it has been processed, and TOP if it hasn't. Then, for a diagram G, we define $chains(G)$ as the set of all Hoare chains

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

for which there exist a list $[\sigma_0, \ldots, \sigma_k]$ of proof-states and a bijection $\pi : k \to V_G \uplus E_G$ with the following properties. First, for all $(\mathbf{v}, \chi, \mathbf{w}) \in E_G$,

$$\begin{array}{ll} \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) < \pi^{-1}(w) & \text{for all } w \in \mathbf{w} \\ \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) & \text{for all } v \in \mathbf{v}. \end{array}$$

Second,

$$\sigma_0 = (initials\, G) \times \{\text{TOP}\}$$

and, for all $i \in k$,

$$\sigma_{i+1} = \begin{cases} (\sigma_i - \{(v, \text{TOP})\}) \uplus \{(v, \text{BOT})\} & \text{if } \pi(i) = v \\ (\sigma_i - (\mathbf{v} \times \{\text{BOT}\})) \uplus (\mathbf{w} \times \{\text{TOP}\}) & \text{if } \pi(i) = (\mathbf{v}, \chi, \mathbf{w}) \end{cases}$$

Third, for all $i \in k + 1$,

$$P_i = \left(\otimes_{(v,\text{TOP}) \in \sigma_i}\, top(\Lambda_G\, v)\right) \otimes \left(\otimes_{(v,\text{BOT}) \in \sigma_i}\, bot(\Lambda_G\, v)\right).$$

Finally, for all $i \in k$,

$$x_i = \begin{cases} \Lambda_G\, v & \text{if } \pi(i) = v \\ \chi & \text{if } \pi(i) = (\mathbf{v}, \chi, \mathbf{w}). \end{cases}$$

Because the '$-$' and '$\uplus$' operators are only partial, we require the following lemma to confirm that the list $[\sigma_0, \ldots, \sigma_k]$ in the above definition is well-defined.

Lemma A.18 (Well-definedness of chains)**.** *For any diagram G, every Hoare chain in $chains(G)$ is well-defined, begins with $top(G)$ and ends with $bot(G)$.*

Our strategy for proving this lemma is mathematical induction on the size of G. First we must modify Defn. A.17, as follows.

Definition A.19 (Extracting Hoare chains – amended)**.** For a diagram G and a set $S \subseteq initials(G)$, we define $chains(G, S)$ as the set of all Hoare chains

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

for which there exist a list $[\sigma_0, \ldots, \sigma_k]$ of proof-states and a bijection $\pi : k \to (V_G \setminus S) \uplus E_G$ with the following properties. (The role of S is to contain those of G's initial nodes which have already been processed, and hence should not be included in the resultant Hoare chains.) First, for all $(\mathbf{v}, \chi, \mathbf{w}) \in E_G$:

$$\begin{array}{ll} \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) < \pi^{-1}(w) & \text{for all } w \in \mathbf{w} \\ \pi^{-1}(v) < \pi^{-1}(\mathbf{v}, \chi, \mathbf{w}) & \text{for all } v \in \mathbf{v} \setminus S. \end{array}$$

Second,

$$\sigma_0 = \{(v, \text{TOP}) \mid v \in (\mathit{initials}\, G) \setminus S\} \cup \{(v, \text{BOT}) \mid v \in S\}.$$

and, for all $i \in k$,

$$\sigma_{i+1} = \begin{cases} (\sigma_i - \{(v, \text{TOP})\}) \uplus \{(v, \text{BOT})\} & \text{if } \pi(i) = v \\ (\sigma_i - (\mathbf{v} \times \{\text{BOT}\})) \uplus (\mathbf{w} \times \{\text{TOP}\}) & \text{if } \pi(i) = (\mathbf{v}, \chi, \mathbf{w}) \end{cases}$$

Third, for all $i \in k+1$,

$$P_i = \left(\otimes_{(v,\text{TOP}) \in \sigma_i}\, \mathit{top}(\Lambda_G\, v)\right) \otimes \left(\otimes_{(v,\text{BOT}) \in \sigma_i}\, \mathit{bot}(\Lambda_G\, v)\right).$$

Finally, for all $i \in k$,

$$x_i = \begin{cases} \Lambda_G\, v & \text{if } \pi(i) = v \\ \chi & \text{if } \pi(i) = (\mathbf{v}, \chi, \mathbf{w}). \end{cases}$$

Lemma A.20. *For all k:*

$$\begin{array}{l} \forall G.\, \forall S \subseteq \mathit{initials}(G).\, \forall H \in \mathit{chains}(G, S). \\ \textit{if } |V_G \setminus S| + |E_G| = k \textit{ then} \\ H \textit{ is well-defined and ends with } \mathit{bot}(G). \end{array}$$

Proof. We use mathematical induction on k.

Case 0**.** Each chain is of the form $\{P_0\}$, so is trivially well-defined. There being no edges, every node is both initial and terminal. We have $V_G = \mathit{initials}(G) = \mathit{terminals}(G) = S$. So

$$\sigma_0 = (\mathit{terminals}\, G) \times \{\text{BOT}\},$$

and hence

$$P_0 = \otimes_{v \in \mathit{terminals}\, G}\, \mathit{bot}(\Lambda_G\, v) = \mathit{bot}(G)$$

as required.

Case $k+1$**.** Each chain is of the form

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}$$

We case-split on whether x_0 is an AsnGadget or a ComGadget.

Case $x_0 \in$ AsnGadget**.** Hence $\pi(0) = v$. Since $[x_0, \ldots, x_{k-1}]$ is a valid linear extension of G, we have $v \in \mathit{initials}(G)$. Since S is excluded from π's co-domain, we have $v \notin S$. Hence σ_0 contains (v, TOP) but not (v, BOT). This ensures that σ_1 is well-defined, and hence, so is the initial step $\{P_0\}\, x_0\, \{P_1\}$ of the chain. It now suffices to show that the rest of the chain is in $\mathit{chains}(G, S \uplus \{v\})$, for then, by the induction hypothesis, the remainder – and hence the entire chain – is well-defined and ends with $\mathit{bot}(G)$. To see this, define a new bijection π' such that $\pi(i) = \pi(i+1)$ for all $i \in k$, and a new list $[\sigma'_0, \ldots, \sigma'_{k-1}] = [\sigma_1, \ldots \sigma_{k+1}]$ and confirm that the four properties listed in Defn. A.19 hold.

Case $x_0 \in \mathsf{ComGadget}$**.** Hence $\pi(0) = (\mathbf{v}, \chi, \mathbf{w})$. Since $[x_0, \ldots, x_{k-1}]$ is a valid linear extension of G, x_0 has no dependants; that is, $\mathbf{v} \subseteq S$. Hence $\mathbf{v} \times \{\text{BOT}\} \subseteq \sigma_0$. Moreover, $\mathbf{w} \times \{\text{BOT}\} \not\parallel \sigma_0$ because $\mathbf{w} \not\parallel (\mathit{initials}\, G)$, which follows from Defn. 5.10. This ensures that σ_1 is well-defined, and hence, so is the initial step of the chain. Consider the graph G' obtained by removing from G the hyperedge $(\mathbf{v}, \chi, \mathbf{w})$ and the vertices in $\mathbf{v}$ (which, by LINEARITY, are not endpoints of any remaining hyperedge). The removal preserves ACYCLICITY and LINEARITY, so G' is well-formed; moreover, $\mathit{bot}(G') = \mathit{bot}(G)$. Let S' be $S \setminus \mathbf{v}$, and note that $|V_{G'} \setminus S'| + |E_{G'}| = k$. The rest of the chain is in $\mathit{chains}(G', S')$, and hence, by the induction hypothesis, is well-defined and ends in $\mathit{bot}(G')$. Thus, the entire chain is well-defined and ends in $\mathit{bot}(G)$. ⁂

Proof of Lemma A.18. It is a straightforward consequence of the definition of σ_0 and Defn. 5.10 that every Hoare chain in $\mathit{chains}(G)$ begins with $\mathit{top}(G)$. We note that when S is empty, $\mathit{chains}(G, S)$ coincides with $\mathit{chains}(G)$, so Lem. A.20 implies the result. ⁂

Proof of Thm. 5.15. We prove the following three statements by mutual rule induction.

$$\vdash^{\text{dia}}_{\mathsf{VaR}} G : P \to Q \Longrightarrow \forall c \in \mathit{coms}\, G. \vdash_{\mathsf{VaR}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\}$$
$$\vdash^{\text{com}}_{\mathsf{VaR}} \chi : P \to Q \Longrightarrow \forall c \in \mathit{coms}\, \chi. \vdash_{\mathsf{VaR}} \{\mathit{asn}\, P\}\, c\, \{\mathit{asn}\, Q\}$$
$$\vdash^{\text{asn}}_{\mathsf{VaR}} A \Longrightarrow \forall c \in \mathit{coms}\, A. \vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}\, A)\}\, c\, \{\mathit{asn}(\mathit{bot}\, A)\}$$

We focus on the MAIN rule, as the others are straightforward consequences of the corresponding separation logic proof rules. Our inductive hypotheses are:

$$\forall v \in V_G. \forall c \in \mathit{coms}(\Lambda_G\, v). \vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}(\Lambda_G\, v))\}\, c\, \{\mathit{asn}(\mathit{bot}(\Lambda_G\, v))\} \tag{A.15}$$
$$\forall (\mathbf{v}, \chi, \mathbf{w}) \in E_G. \forall c \in \mathit{coms}\, \chi. \vdash_{\mathsf{VaR}} \{\mathit{asn}(\otimes_{v \in \mathbf{v}}\, \mathit{bot}(\Lambda_G\, v))\}\, c\, \{\mathit{asn}(\otimes_{w \in \mathbf{w}}\, \mathit{top}(\Lambda_G\, w))\} \tag{A.16}$$

We are to prove, for all $c \in \mathit{coms}\, G$, that:

$$\vdash_{\mathsf{VaR}} \{\mathit{asn}(\mathit{top}\, G)\}\, c\, \{\mathit{asn}(\mathit{bot}\, G)\}. \tag{A.17}$$

Observe that c can be written c_0 ; $\cdots$; c_{k-1} ; `skip` for some linear extension $[x_0, \ldots, x_{k-1}]$ of G, where $c_i \in \mathit{coms}(x_i)$ for all $i \in k$. Consider the corresponding Hoare chain:

$$\{P_0\}\, x_0\, \{P_1\}\, \cdots\, \{P_{k-1}\}\, x_{k-1} \{P_k\}.$$

We pick an arbitrary $i \in k$, and proceed depending on whether x_i is an AsnGadget or a ComGadget.

Case $x_i \in \mathsf{AsnGadget}$**.** Hence $\pi(i) = v$. Hence

$$\sigma_{i+1} = (\sigma_i - \{(v, \text{TOP})\}) \uplus \{(v, \text{BOT})\}.$$

By Lem. A.18, this expression is well-defined, and hence

$$\begin{aligned} \sigma_i &= \{(v, \text{TOP})\} \uplus \sigma' \\ \sigma_{i+1} &= \{(v, \text{BOT})\} \uplus \sigma' \end{aligned}$$

for some σ'. Hence

$$\begin{aligned} P_i &= \mathit{top}(\Lambda_G\, v) \otimes P' \\ P_{i+1} &= \mathit{bot}(\Lambda_G\, v) \otimes P' \end{aligned}$$

for some P'. By (A.15), we have

$$\vdash_{\mathsf{VaR}} \{asn(top(\Lambda_G\, v))\}\; c_i\; \{asn(bot(\Lambda_G\, v))\}$$

from which

$$\vdash_{\mathsf{VaR}} \{asn(top(\Lambda_G\, v)) * asn\; P'\}\; c_i\; \{asn(bot(\Lambda_G\, v)) * asn\; P'\}$$

follows by separation logic's frame rule (which, under variables-as-resource, has no side-conditions). Hence

$$\vdash_{\mathsf{VaR}} \{asn\; P_i\}\; c_i\; \{asn\; P_{i+1}\}.$$

Case $x_i \in \mathsf{ComGadget}$**.** Similar, using (A.16) instead of (A.15) .

We then use Hoare logic's sequencing rule to assemble a proof of the entire chain:

$$\vdash_{\mathsf{VaR}} \{asn\; P_0\}\; c\; \{asn\; P_k\}.$$

It remains to show that P_0 is $top(G)$ and P_k is $bot(G)$; this follows directly from Lem. A.18.

Bibliography

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Levy. Explicit substitutions. In Frances E. Allen, editor, *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*, pages 31–46. ACM Press, 1990. (Cited on page 49.)

Hasan Amjad and Richard Bornat. Towards automatic stability analysis for rely-guarantee proofs. In Neil D. Jones and Markus Müller-Olm, editors, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '09)*, volume 5403 of *Lecture Notes in Computer Science*, pages 14–28. Springer-Verlag, 2009. (Cited on page 49.)

Edward A. Ashcroft. Program verification tableaus. Technical Report CS-76-01, University of Waterloo, 1976. (Cited on pages 97 and 126.)

Jules Bean. Ribbon proofs. In *Proceedings of the 19th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XIX)*, volume 83 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005. See also Bean [2006]. (Cited on pages 97 and 99.)

Jules Bean. *Ribbon Proofs - A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006. (Cited on page 143.)

Bell Labs. Version 7 Unix: malloc. Source code, 1979. URL `http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/libc/gen/malloc.c`. (Cited on pages 14, 63, and 83.)

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO '05)*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. (Cited on pages 97 and 100.)

Richard Bornat and Mike Dodds. Abducing memory barriers. Draft, February 2012. URL `http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/abducingbarriers.pdf`. (Cited on pages 97, 99, and 127.)

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 259–270. ACM Press, 2005. (Cited on pages 33 and 97.)

Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in separation logic. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier, 2006. (Cited on pages 18, 33, and 110.)

Stephen Brookes. A semantics for concurrent separation logic. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR '04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004. See also Brookes [2007]. (Cited on pages 18, 37, 91, and 127.)

Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. (Cited on page 144.)

Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *Proceedings of the 14th International Symposium on Static Analysis (SAS '07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2007. (Cited on page 49.)

Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 234–245. ACM Press, 2011. (Cited on page 125.)

Maurice Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973. (Cited on page 43.)

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. (Cited on page 43.)

Joey W. Coleman. Expression decomposition in a rely/guarantee context. In Natarajan Shankar and Jim Woodcock, editors, *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE '08)*, volume 5295 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2008. (Cited on pages 49 and 54.)

Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007. (Cited on page 49.)

Byron Cook, Jasmin Fisher, Elzbieta Krepska, and Nir Piterman. Proving stabilization of biological systems. In Ranjit Jhala and David A. Schmidt, editors, *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2011. (Cited on page 93.)

Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1976. (Cited on page 51.)

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP '10)*, volume 6183 of

Lecture Notes in Computer Science, pages 504–528. Springer, 2010. (Cited on pages 18, 44, 92, 97, 99, and 127.)

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In Giuseppe Castagna, editor, *Proceedings of the 18th European Symposium on Programming (ESOP '09)*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009. (Cited on pages 18, 92, 93, 97, 99, and 127.)

Xinyu Feng. Local rely-guarantee reasoning. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, pages 315–327. ACM Press, 2009. (Cited on pages 18, 91, 92, 97, 99, and 127.)

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP '07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007. (Cited on pages 18, 44, 97, 99, and 127.)

Charles Fishman. They write the right stuff. *Fast Company*, 6, December 1996. (Cited on page 14.)

Frederick B. Fitch. *Symbolic Logic: An Introduction*. Ronald Press Co., 1952. (Cited on page 99.)

Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3):153–183, 2005. (Cited on page 55.)

Robert W. Floyd. Assigning meanings to programs. In Jacob T. Schwartz, editor, *Proceedings of the American Mathematical Society Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967. (Cited on page 58.)

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. (Cited on page 126.)

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. (Cited on page 67.)

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In Zhong Shao, editor, *Proceedings of the 5th Asian Symposium on Programming Languages and Systems (APLAS '07)*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2007. (Cited on pages 97, 99, and 127.)

Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, pages 16–28. ACM Press, 2009. (Cited on page 93.)

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on pages 13, 18, and 19.)

C. A. R. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, 1971a. (Cited on page 97.)

C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971b. (Cited on page 18.)

Gerard J. Holzmann. Economics of software verification. In John Field and Gregor Snelting, editors, *Proceedings of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE '01)*. ACM Press, 2001. (Cited on page 14.)

Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985. (Cited on page 13.)

Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Separation logic in the presence of garbage collection. In Martin Grohe, editor, *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS '11)*, pages 247–256. IEEE Computer Society, 2011. (Cited on pages 97, 99, and 127.)

Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26. ACM Press, 2001. (Cited on pages 18, 28, 97, and 99.)

Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, pages 271–282. ACM Press, 2011. (Cited on page 43.)

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods – Proceedings of the 3rd International Symposium (NFM '11)*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011a. (Cited on pages 43 and 125.)

Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules. In Michael Butler and Wolfram Schulte, editors, *Proceedings of the 17th International Symposium on Formal Methods (FM '11)*, volume 6664 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2011b. (Cited on pages 97, 99, and 127.)

Jonas B. Jensen and Lars Birkedal. Fictional separation logic. In Helmut Seidl, editor, *Proceedings of the 21st European Symposium on Programming (ESOP '12)*, volume 7211 of *Lecture Notes in Computer Science*, pages 377–396. Springer, 2012. (Cited on page 92.)

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. (Cited on pages 18, 36, 37, and 39.)

Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, Inc., second edition, 1990. (Cited on pages 21 and 129.)

Cliff B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 167–187. Springer, 2010. (Cited on page 43.)

André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996. (Cited on page 127.)

Brian W. Kernighan and M. Douglas McIlroy. *Unix time-sharing system: Unix programmer's manual*. Bell Telephone Laboratories, Inc., seventh edition, 1979. (Cited on page 65.)

Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In Peter Thieman and Robby Bruce Findler, editors, *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*, pages 41–54. ACM Press, 2012. (Cited on page 92.)

Philippe Lacan, Jean Noël Monfort, Le Vinh Quy Ribal, Alain Deutsch, and Georges Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings of the Conference on Data Systems in Aerospace (DASIA '98)*, pages 201–205. European Space Agency, 1998. (Cited on page 13.)

John McCarthy and James A. Painter. Correctness of a compiler for arithmetical expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 33–41. AMS, 1967. (Cited on page 13.)

Steve McConnell. *Code Complete*. Microsoft Press, 2nd edition, 2004. (Cited on page 13.)

Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009. (Cited on page 126.)

Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981. (Cited on pages 91 and 97.)

Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. Wiley, third edition, 2012. (Cited on page 13.)

NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report. National Institute of Standards and Technology, U.S. Department of Commerce, May 2002. (Cited on page 13.)

Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR '04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004. See also O'Hearn [2007]. (Cited on pages 18, 37, 38, 91, and 127.)

Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. (Cited on pages 99 and 147.)

Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. (Cited on page 99.)

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 268–280. ACM Press, 2004. See also O'Hearn et al. [2009]. (Cited on pages 15, 25, 63, 64, 66, and 91.)

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. *ACM Transactions on Programming Languages and Systems*, 31(3), 2009. (Cited on page 147.)

Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. (Cited on pages 18, 36, 43, and 97.)

Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 247–258. ACM Press, 2005. (Cited on pages 15, 18, 25, 64, 65, 66, and 93.)

Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In Rajeev Alur, editor, *Proceedings of the 21st IEEE Symposium on Logic in Computer Science (LICS '06)*, pages 137–146. IEEE Computer Society, 2006. (Cited on page 34.)

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. Modular verification of a non-blocking stack. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, pages 297–302. ACM Press, 2007. (Cited on pages 37 and 43.)

Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP '03)*, volume 2618 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2003. (Cited on pages 39 and 49.)

Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. Automatic parallelization with separation logic. In Giuseppe Castagna, editor, *Proceedings of the 18th European Symposium on Programming (ESOP '09)*, volume 5502 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2009. (Cited on page 127.)

John C. Reynolds. Reasoning about arrays. *Communications of the ACM*, 22(5):290–299, 1979. (Cited on page 32.)

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In Gordon D. Plotkin, editor, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, 2002. (Cited on pages 18, 28, 97, and 103.)

Fred B. Schneider. *On Concurrent Programming*, chapter 4. Springer, 1997. (Cited on pages 97 and 126.)

Peter Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, volume 813, chapter 4. Springer, 2011. (Cited on page 127.)

Robert D. Tennent. *Specifying software*. Cambridge University Press, 2002. (Cited on page 13.)

Matej Urbas and Mateja Jamnik. Diabelli: A heterogeneous proof system. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 559–566. Springer, 2012. (Cited on page 126.)

Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, July 2007. (Cited on pages 14, 47, 50, 59, 60, 63, 69, and 91.)

Viktor Vafeiadis. RGSep action inference. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '10)*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer, 2010. (Cited on page 92.)

Viktor Vafeiadis. Concurrent separation logic and operational semantics. In Joël Ouaknine, editor, *Proceedings of the 27th Annual Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII)*, volume 276 of *Electronic Notes in Theoretical Computer Science*, pages 335–351. Elsevier, 2011. (Cited on pages 20 and 38.)

Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR '07)*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. (Cited on pages 18, 43, 45, 69, 97, 99, and 127.)

Makarius Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In David Aspinall and Claudio Sacerdoti Coen, editors, *Proceedings of the 9th International Workshop on User Interfaces for Theorem Provers (UITP '10)*, volume 285 of *Electronic Notes in Theoretical Computer Science*, pages 101–114. Elsevier, 2012. (Cited on page 125.)

John Wickerson, Mike Dodds, and Matthew J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In Andrew D. Gordon, editor, *Proceedings of the 19th European Symposium on Programming (ESOP '10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 610–629. Springer, 2010a. (Cited on pages 49 and 63.)

John Wickerson, Mike Dodds, and Matthew J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. Technical Report UCAM-CL-TR-774, University of Cambridge, 2010b. (Cited on page 90.)

John Wickerson, Mike Dodds, and Matthew J. Parkinson. Ribbon proofs for separation logic. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, volume 7792 of *Lecture Notes in Computer Science*, pages 189–208. Springer, 2013. (Cited on page 97.)

Chunhan Wu, Xingyuan Zhang, and Christian Urban. A formalisation of the Myhill-Nerode theorem based on regular expressions. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP '11)*, volume 6898 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2011. (Cited on page 111.)

Greta Yorsh, Alexey Skidanov, Thomas Reps, and Mooly Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs (ongoing work). In Agostino Cortesi and Francesco Logozzo, editors, *Proceedings of the 1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL '05)*, volume 131 of *Electronic Notes in Theoretical Computer Science*, pages 125–138. Elsevier, 2005. (Cited on page 91.)