Technical Report

UCAM-CL-TR-758
ISSN 1476-2986

Number 758

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Logics and analyses for concurrent heap-manipulating programs

Alexey Gotsman

October 2009

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 September 2009 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

ISSN 1476-2986

Abstract

Reasoning about concurrent programs is difficult because of the need to consider all possible interactions between concurrently executing threads. The problem is especially acute for programs that manipulate shared heap-allocated data structures, since heap-manipulation provides more ways for threads to interact. Modular reasoning techniques sidestep this difficulty by considering every thread in isolation under some assumptions on its environment.

In this dissertation we develop modular program logics and program analyses for the verification of concurrent heap-manipulating programs. Our approach is to exploit reasoning principles provided by program logics to construct modular program analyses and to use this process to obtain further insights into the logics. In particular, we build on concurrent separation logic—a Hoare-style logic that allows modular manual reasoning about concurrent programs written in a simple heap-manipulating programming language.

Our first contribution is to show the soundness of concurrent separation logic without the conjunction rule and the restriction that resource invariants be precise, and to construct an analysis for concurrent heap-manipulating programs that exploits this modified reasoning principle to achieve modularity. The analysis can be used to automatically verify a number of safety properties, including memory safety, data-structure integrity, data-race freedom, the absence of memory leaks, and the absence of assertion violations. We show that we can view the analysis as generating proofs in our variant of the logic, which enables the use of its results in proof-carrying code or theorem proving systems.

Reasoning principles expressed by program logics are most often formulated for only idealised programming constructs. Our second contribution is to develop logics and analyses for modular reasoning about features present in modern languages and libraries for concurrent programming: storable locks (i.e., locks dynamically created and destroyed in the heap), first-order procedures, and dynamically-created threads.

Acknowledgments

‘So, what do you think?’
‘It’s wrong.’
‘Do you mean it’s incorrect?!’
‘No, it’s correct, but it’s wrong.’

A conversation with Josh Berdine

I would first like to thank Anatoly Ligun, who got me interested in research back in my high-school years, Oleg Revin, whose excellent undergraduate course in logic determined my research area, and Alexander Khizha, who supervised my first steps in it.

I am grateful to my (both formal and informal) advisors, Josh Berdine, Byron Cook, and Mike Gordon, for their guidance, assistance, and encouragement during the course of my research. I am also grateful to Noam Rinetzky and Mooly Sagiv, with whom I coauthored some of the papers this dissertation is based on.

I would like to thank the Cambridge Overseas Trust for awarding me a TNK-BP Kapitza Cambridge Scholarship that made it possible for me to study at Cambridge.

I am grateful to the following people for useful discussions and comments about this dissertation and the papers it is based on: Richard Bornat, Cristiano Calcagno, Dino Distefano, Tal Lev-Ami, Stephen Magill, Roman Manevich, Alan Mycroft, Peter O’Hearn, Matthew Parkinson, Andreas Podelski, Zvonimir Rakamaric, Ganesan Ramalingam, John Reynolds, Peter Sewell, Viktor Vafeiadis, Hongseok Yang, Greta Yorsh, and Jian Zhang.

Finally, I would like to thank my family for all their support.

Contents

Chapter 1

Introduction

This dissertation is about program verification, which is concerned with formally proving desirable properties of computer programs. Of the spectrum of the currently available approaches to verification, here we are interested in two: program logics and program analyses. Program logics are proof systems that formalise reasoning principles for arguing program correctness. They are most often used in manual proofs, which require insight into the reasons for the program being correct. Program analyses are algorithms that compute an over-approximation of the denotation of the program, and are implemented as automatic tools. They usually use the simplest reasoning principles, compensating for this with brute force. For example, analyses based on abstract interpretation typically involve an iterative computation accumulating facts (expressed in so-called abstract domains) that describe possible states the program might visit. It is rare to see a subtle reasoning principle of a program logic being applied in a program analysis, so that instead of computing the denotation of the program directly, the analysis computes an incarnation of the reasoning principle. In this dissertation, we develop logics and analyses exhibiting such a connection for a class of programs, concurrent heap-manipulating programs, for which analyses based on straightforward reasoning principles are infeasible.

A straightforward way of reasoning about a concurrent program is by considering all interleavings of executions of its threads. The number of possible interleavings tends to be huge, thus, this approach results in program analyses being unscalable and proofs in program logics being intractable. The problem is especially acute for programs that manipulate shared heap-allocated data structures, since heap-manipulation provides more ways for concurrently executing threads to interact. One way to overcome this problem is to use techniques for modular reasoning, i.e., those which consider every thread or component in the program in isolation under some assumptions on its environment and thus avoid reasoning about all thread interactions directly. Modular reasoning techniques for shared-variable concurrent programs typically partition program variables into thread-local and shared; assumptions on a thread's environment need only specify how the environment changes the shared variables. For heap-manipulating programs, such a priori fixed partitioning into thread-local and shared state is usually inappropriate, which makes devising modular techniques for them challenging.

Striking progress in this realm has recently been made by O'Hearn [56], who proposed concurrent separation logic as a basis for reasoning about such programs. The main idea of the reasoning principle provided by the logic is to partition the heap into thread-local and shared parts while allowing the partitioning to change: the logic permits the ownership of memory cells to be transferred from thread-local parts of the heap into shared and back. Interaction between threads via shared parts is mediated using resource invariants, which are assertions about them that must be respected by every thread. This is supported by a mechanism for modular sequential reasoning using the separating conjunction connective. The reasoning method based on dynamic heap splitting is subtle: the logic is unsound unless resource invariants satisfy the restriction of precision, informally requiring that they unambiguously carve out an area of the heap.

Our first contribution is to show the soundness of concurrent separation logic without this restriction on resource invariants and one of the rules (the conjunction rule), and to construct an analysis for concurrent heap-manipulating programs that exploits this modified reasoning principle to achieve modularity. Namely, our analysis infers (generally imprecise) resource invariants thread-modularly, i.e., by repeatedly considering each individual thread instead of the whole program and thus avoiding interleaving enumeration. It can be used to automatically verify a number of safety properties, including memory safety, data-structure integrity, data-race freedom, the absence of memory leaks, and the absence of assertion violations.[1] We prove the soundness of our variant of concurrent separation logic and the analysis together, in a uniform framework. Furthermore, we show that we can view the analysis as generating proofs in our variant of the logic, which enables the use of its results in proof-carrying code or theorem proving systems. In this case, trying to implement a reasoning principle of a program logic in a program analysis leads to an additional insight into the logic, suggesting an alternative way of attaining soundness.

Reasoning principles expressed by program logics are most often formulated for only idealised programming constructs. For example, concurrent separation logic essentially reasons about programs with static locks (i.e., locks declared as global variables) and threads created in a well-structured manner using parallel composition. Our second contribution is to develop logics and analyses for modular reasoning about features present in modern languages and libraries for concurrent programming [71, 6, 44]: storable locks (i.e., locks dynamically created and destroyed in the heap), first-order procedures, and dynamically-created threads. In some cases, this requires us to resolve set-theoretic paradoxes in the models of the logics. As before, the principles of reasoning about the corre-

[1] In this dissertation we restrict ourselves to verifying safety properties. Developing methods for verifying liveness properties of concurrent heap-manipulating programs has been the subject of our ongoing work [35, 18].

sponding language constructs used by the logics we develop for manual reasoning and by our analyses differ: the latter may invalidate the conjunction rule in exchange for lifting the restrictions imposed by the former that are hard for the analyses to satisfy.

1.1 Dissertation outline

In Chapter 2 we present the necessary technical background from program logics and program analyses, providing a brief overview of concurrent separation logic and abstract interpretation.

In Chapter 3 we consider a simple concurrent programming language in which programs consist of a top-level parallel composition of threads synchronising via static locks. We first present a novel framework for constructing thread-modular analyses for such programs out of analyses for the sequential subset of the language that operate on abstract separation domains, i.e., abstract domains with a separating conjunction-like operation defined on them. We give several examples of abstract separation domains and present an instantiation of the framework with a sequential heap analysis based on separation logic. We further demonstrate that our analysis can be viewed as generating proofs in a variant of concurrent separation logic without the conjunction rule and the restriction that resource invariants be precise. We prove the soundness of the analysis and both the standard and our variants of the logic, thereby resolving the open question about the soundness of concurrent separation logic without the conjunction rule and the precision restriction.

In Chapter 4 we extend the programming language with commands for allocation and deallocation of locks in the heap. We design a new logic and a corresponding analysis that treat storable locks along with the data structures they protect as resources, assigning invariants to them and managing their dynamic creation and destruction. This task is nontrivial, as the straightforward definition of a resource-oriented model for storable locks leads to a form of Russell's paradox, circularity arising from locks referring to themselves through their resource invariants. We address this foundational difficulty by cutting the circularity with an indirection, which lets us give a simple semantics to our logic and yields the formulation of an appropriate analysis. We demonstrate that the proposed logic allows modular reasoning about programs for which there exists a notion of dynamic ownership of heap parts by locks and threads. The class of such programs contains programs with coarse-grained synchronisation and some, but not all, programs with fine-grained synchronisation, including examples that were posed as challenges in the literature. As in Chapter 3, we consider two variants of the logic: one that requires resource invariants to be precise, but includes the conjunction rule, and one that omits the rule and the restriction.

In Chapter 5 we consider concurrent programs with first-order procedures. Interprocedural program analysis of heap-manipulating programs is challenging even in the sequential setting because of the explosion in the number of calling contexts that the analysis has to consider. One way to combat this problem is to perform localisation at procedure calls in the analysis (i.e., to pass only the relevant part of the program state to the procedure), which usually reduces the number of contexts. We develop a framework for constructing interprocedural analyses with localisation for sequential programs out of intraprocedural analyses operating on abstract separation domains that generalises the Reps-Horwitz-Sagiv algorithm for interprocedural analysis. This extends the existing work on interprocedural analysis of sequential heap-manipulating programs, which so far has considered only particular abstract domains or localisation schemes. We also present an instantiation of our framework with an abstract domain based on separation logic. We propose an abstract (i.e., interpreted over a class of models) version of separation logic for programs with procedures that our analysis generates proofs in. We give two proofs of soundness to the logic. The first one is an elegant proof using standard techniques that establishes the soundness of the whole logic, including the conjunction rule, by computing the best predicate transformer corresponding to the procedure specifications used in the proof of a program. The second is a novel proof that avoids computing the transformer at the price of being more complicated and not establishing the soundness of the conjunction rule. However, unlike the former proof, this one can be adapted to the concurrent setting even when resource invariants may be imprecise and the conjunction rule does not hold. This allows us to show that our interprocedural analysis can soundly be composed with the thread-modular analysis of Chapter 3.

In Chapter 6 we move from a top-level parallel composition of threads to dynamic thread creation, adding a command for forking a new thread to the programming language of Chapter 4. We present a logic for dynamic thread creation and construct a corresponding analysis using the methods proposed in the previous chapters. We thus demonstrate that these methods can be applied to language constructs other than the ones they were originally proposed for.

In Chapter 7 we conclude and note some directions of further research that our results suggest.

Collaboration. The thread-modular heap analysis of Chapter 3 first appeared in the PLDI'07 paper *Thread-modular shape analysis*, coauthored with Josh Berdine, Byron Cook, and Mooly Sagiv [34]. The logic for storable locks of Chapter 4 in the case of precise resource invariants and a variant of the logic for threads of Chapter 6 were proposed in the APLAS'07 paper and a companion technical report *Local reasoning for storable locks and threads*, coauthored with Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv [32, 33]. A version of the interprocedural heap analysis for sequential programs of Chapter 5 with an abstract domain based on separation logic was published in the SAS'06 paper *Interprocedural shape analysis with separated heap abstractions*, coauthored with Josh Berdine and Byron Cook [31].

Follow-on work. In the time between the above publications and the submission of this dissertation, a number of other papers addressing the topics discussed here have appeared, some of them following on our results. We discuss them in the related work sections of the corresponding chapters.

Chapter 2

Technical background

2.1 Separation logic

In this dissertation, by program logics we mean Hoare logics [41]. In particular, we are interested in a Hoare logic for heap-manipulating programs—separation logic [45, 66]. We now review the foundations of the version of separation logic for sequential and concurrent programs proposed by Calcagno et al. [15]. This version of the logic is abstract in the sense that it can be interpreted over a wide class of semantic models with a given structure, which allows reusing results about the logic in multiple contexts. As any Hoare logic, separation logic includes two formal systems—one for assertions (formulae describing program states) and one for Hoare triples (logical judgements describing the effect of commands on program states). We discuss the former first.

2.1.1 Assertions

Separation algebras and domains. In abstract separation logic, assertions are interpreted with respect to a separation algebra, which represents program states.

Definition 2.1 (Separation algebra). *A separation algebra is a partial commutative semigroup* $(\Sigma, *)$. *A partial commutative semigroup is given by a partial binary operation of separate combination* $*$ *where the commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined:*

$$\forall \sigma_1, \sigma_2, \sigma_3 \in \Sigma.\, \sigma_1 * (\sigma_2 * \sigma_3) = (\sigma_1 * \sigma_2) * \sigma_3;$$
$$\forall \sigma_1, \sigma_2 \in \Sigma.\, \sigma_1 * \sigma_2 = \sigma_2 * \sigma_1.$$

The original definition of separation algebras given in [15] requires a separation algebra to have a unit element and the $*$ operation to be *cancellative.* The latter requirement states that for each $\sigma \in \Sigma$, the partial function $\sigma * \cdot : \Sigma \rightharpoonup \Sigma$ is injective, and is connected with conditions for validating the conjunction rule of Hoare logic. Since in this dissertation we also consider models of concurrent separation logic invalidating the conjunction rule, we omit this requirement.

In this dissertation, by a *domain* D we understand a join-semilattice $(D, \sqsubseteq, \bigsqcup, \bot, \top)$ with a bottom element $\bot$. For a set Σ let $\mathcal{P}(\Sigma)^\top$ be the domain of subsets of Σ with a special element $\top$. The order $\sqsubseteq$ in the domain $\mathcal{P}(\Sigma)^\top$ is subset inclusion with $\top$ being the greatest element. When Σ represents program states, we usually use $\top$ to denote an error state resulting, e.g., from dereferencing an invalid pointer. Note that the order $\sqsubseteq$ defines the corresponding join $\sqcup$ and meet $\sqcap$ operations on the domain $\mathcal{P}(\Sigma)^\top$. If Σ is a separation algebra, we can lift the $*$ operation to $\mathcal{P}(\Sigma)^\top$ pointwise: for all $p, q \in \mathcal{P}(\Sigma)$

$$p * q = \bigcup\{\sigma * \eta \mid \sigma \in p, \eta \in q, \sigma * \eta \text{ is defined}\}; \quad \top * p = p * \top = \top.$$

Thus, $\mathcal{P}(\Sigma)^\top$ has a total commutative semigroup structure. We can generalise this construction to the notion of an arbitrary separation domain.

Definition 2.2 (Separation domain). *A separation domain is a domain* $(D, \sqsubseteq, \bigsqcup, \bot, \top, *, e)$ *equipped with an operation of separate combination* $* : (D \times D) \to D$ *such that* $(D, \sqsubseteq, *, e)$ *is a partially-ordered commutative monoid, i.e.,*

- $*$ *is associative and commutative:*

$$\forall p_1, p_2, p_3 \in D.\, p_1 * (p_2 * p_3) = (p_1 * p_2) * p_3;$$
$$\forall p_1, p_2 \in D.\, p_1 * p_2 = p_2 * p_1;$$

- $*$ *has the unit* e*:* $\forall p \in D.\, p * e = p;$

- $*$ *is monotone:* $\forall p_1, p_2, q \in D.\, p_1 \sqsubseteq p_2 \Rightarrow p_1 * q \sqsubseteq p_2 * q.$

The requirement that a separation domain must have a unit is imposed here for technical convenience. For a separation algebra Σ in the case when the domain $\mathcal{P}(\Sigma)^\top$ defined above has a unit, we call it the separation domain constructed out of the algebra Σ. We denote with $\circledast$ the iterated version of $*$:

$$\underset{k=1}{\overset{n}{\circledast}} p_k = e * p_1 * \ldots * p_n.$$

In the future, for $\sigma \in \Sigma \cup \{\top\}$ we denote with $\{\!|\sigma|\!\}$ the singleton set containing σ if $\sigma \in \Sigma$ and $\top$ if $\sigma = \top$ (thus, $\{\!|\sigma|\!\} \in \mathcal{P}(\Sigma)^\top$).

Example of a separation domain. Elements of separation algebras and domains are often defined using partial functions. We use the following notation: $f(x)\downarrow$ means that the function f is defined on x, $f(x)\uparrow$ means that the function f is undefined on x, $\mathsf{dom}(f)$ denotes the set of arguments on which the function f is defined, and $[\,]$ denotes a nowhere-defined function. We denote with $f[x : y]$ the function that has the same value

$$\begin{aligned}
&\text{Values} = \{\dots, -1, 0, 1, \dots\}\\
&\text{Perms} = (0, 1]\\
&\text{Locs} = \{1, 2, \dots\}\\
&\text{Vars} = \{\mathtt{x}, \mathtt{y}, \dots\}\\
&\text{LVars} = \{X, Y, \dots\}\\
&\text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms})\\
&\text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \mathbf{Cell}(\text{Values})\\
&\text{Ints} = \text{LVars} \to \text{Values}\\
&\text{States} = \text{Stacks} \times \text{Heaps} \times \text{Ints}
\end{aligned}$$

Figure 2.1: Example of a separation algebra

as f everywhere, except for x, where it has the value y (even if $f(x)\uparrow$). $f \uplus g$ is the union of the disjoint partial functions f and g. It is undefined if $\mathsf{dom}(f) \cap \mathsf{dom}(g) \neq \emptyset$.

Figure 2.1 defines a separation algebra States used to define the semantics of heap-manipulating programs. We distinguish integer program variables Vars (which may appear in programs) and logical variables LVars (which do not appear in programs, only in formulae). A state of the program is a triple of a stack, a heap, and an interpretation. A stack is a finite partial function from variables to values and permissions (numbers from $(0, 1]$ that show "how much" of the variable is owned by the assertion), a heap is a finite partial function from locations to values, and an interpretation is a total function from logical variables to values. In the following chapters, we define algebras with several kinds of heap cells, therefore, in Figure 2.1 we mark ordinary heap cells introduced here with the constructor **Cell**. Logical variables and fractional permissions [10, 7] for program variables are necessary for obtaining a complete (in the sense of [62]) proof system for this separation algebra. For clarity of presentation we omit the treatment of permissions for memory cells.

We define the operation of separate combination on states in the following way. For $s_1, s_2 \in$ Stacks let

$$s_1 \natural s_2 \Leftrightarrow (\forall \mathtt{x}.\, s_1(\mathtt{x})\downarrow \wedge s_2(\mathtt{x})\downarrow \Rightarrow (\exists u, \pi_1, \pi_2.\, s_1(\mathtt{x}) = (u, \pi_1) \wedge s_2(\mathtt{x}) = (u, \pi_2) \wedge \pi_1 + \pi_2 \leq 1)).$$

If $s_1 \natural s_2$, then we define

$$\begin{aligned} s_1 * s_2 = \{(\mathtt{x}, (u, \pi)) \mid\ & (s_1(\mathtt{x}) = (u, \pi) \wedge s_2(\mathtt{x})\uparrow) \vee (s_2(\mathtt{x}) = (u, \pi) \wedge s_1(\mathtt{x})\uparrow) \vee \\ & (s_1(\mathtt{x}) = (u, \pi_1) \wedge s_2(\mathtt{x}) = (u, \pi_2) \wedge \pi = \pi_1 + \pi_2)\}; \end{aligned}$$

otherwise $s_1 * s_2$ is undefined. For $h_1, h_2 \in$ Heaps we define $h_1 * h_2 = h_1 \uplus h_2$. The $*$-combination $\mathbf{i}_1 * \mathbf{i}_2$ of interpretations $\mathbf{i}_1, \mathbf{i}_2 \in$ Ints is defined to be $\mathbf{i}_1$ only if $\mathbf{i}_1 = \mathbf{i}_2$. For $(s_1, h_1, \mathbf{i}_1) \in$ States and $(s_2, h_2, \mathbf{i}_2) \in$ States we then let

$$(s_1, h_1, \mathbf{i}_1) * (s_2, h_2, \mathbf{i}_2) = (s_1 * s_2, h_1 * h_2, \mathbf{i}_1 * \mathbf{i}_2).$$

$$
\begin{array}{lcl}
(s, h, \mathbf{i}) \models E \mapsto F & \Leftrightarrow & [\![E]\!]_{s,\mathbf{i}}\downarrow \wedge [\![F]\!]_{s,\mathbf{i}}\downarrow \wedge h = [[\![E]\!]_{s,\mathbf{i}} : \mathbf{Cell}([\![F]\!]_{s,\mathbf{i}})] \\
(s, h, \mathbf{i}) \models \mathsf{Own}_\pi(\mathtt{x}) & \Leftrightarrow & \exists u.\, [\![\pi]\!]_{s,\mathbf{i}}\downarrow \wedge s = [\mathtt{x} : (u, [\![\pi]\!]_{s,\mathbf{i}})] \wedge 0 < [\![\pi]\!]_{s,\mathbf{i}} \leq 1 \\
(s, h, \mathbf{i}) \models \mathsf{emp}_\mathbf{s} & \Leftrightarrow & s = [\,] \\
(s, h, \mathbf{i}) \models \mathsf{emp}_\mathbf{h} & \Leftrightarrow & h = [\,] \\
(s, h, \mathbf{i}) \models E = F & \Leftrightarrow & [\![E]\!]_{s,\mathbf{i}}\downarrow \wedge [\![F]\!]_{s,\mathbf{i}}\downarrow \wedge [\![E]\!]_{s,\mathbf{i}} = [\![F]\!]_{s,\mathbf{i}} \\
(s, h, \mathbf{i}) \models \pi_1 = \pi_2 & \Leftrightarrow & [\![\pi_1]\!]_{s,\mathbf{i}}\downarrow \wedge [\![\pi_2]\!]_{s,\mathbf{i}}\downarrow \wedge [\![\pi_1]\!]_{s,\mathbf{i}} = [\![\pi_2]\!]_{s,\mathbf{i}} \\
(s, h, \mathbf{i}) \models P \Rightarrow Q & \Leftrightarrow & ((s, h, \mathbf{i}) \models P) \Rightarrow ((s, h, \mathbf{i}) \models Q) \\
(s, h, \mathbf{i}) \models \mathbf{false} & \Leftrightarrow & \mathbf{false} \\
(s, h, \mathbf{i}) \models P * Q & \Leftrightarrow & \\
\multicolumn{3}{l}{\qquad \exists s_1, h_1, s_2, h_2.\, s = s_1 * s_2 \wedge h = h_1 * h_2 \wedge (s_1, h_1, \mathbf{i}) \models P \wedge (s_2, h_2, \mathbf{i}) \models Q} \\
(s, h, \mathbf{i}) \models P \mathrel{-\!\!*} Q & \Leftrightarrow & \\
\multicolumn{3}{l}{\qquad \forall s', h'.\, s \natural s' \wedge h \natural h' \wedge ((s', h', \mathbf{i}) \models P) \Rightarrow ((s * s', h * h', \mathbf{i}) \models Q)} \\
(s, h, \mathbf{i}) \models \exists X.\, P & \Leftrightarrow & \exists u.\, (s, h, \mathbf{i}[X : u]) \models P
\end{array}
$$

Figure 2.2: Satisfaction relation for the assertion language formulae: $(s, h, \mathbf{i}) \models P$

Let RAM be the separation domain constructed out of the algebra States, i.e., $\mathsf{RAM} = \mathcal{P}(\text{States})^\top$. Note that although the separation algebra States does not have a unit, the separation domain RAM has one: $\{[\,]\} \times \{[\,]\} \times \text{Ints}$.

Assertion language. The following syntax defines formulae P, Q of the assertion language interpreted over the domain RAM:

$$
\begin{array}{rcl}
\mathtt{x} & \in & \text{Vars} \\
X & \in & \text{LVars} \\
E, F & ::= & \mathtt{NULL} \mid \mathtt{x} \mid X \mid E + F \mid \ldots \\
P, Q & ::= & \mathbf{false} \mid P \Rightarrow Q \mid \exists X.\, P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathsf{emp}_\mathbf{s} \mid \mathsf{emp}_\mathbf{h} \\
& \mid & E = F \mid \pi_1 = \pi_2 \mid \mathsf{Own}_\pi(\mathtt{x}) \mid E \mapsto F
\end{array}
$$

where π, π_1, π_2 range over permission expressions, evaluating to numbers from $(0, 1]$.

The satisfaction relation for the assertion language formulae is defined in Figure 2.2. We assume a function $[\![E]\!]_{s,\mathbf{i}}$ that evaluates an expression with respect to the stack s and the interpretation $\mathbf{i}$ and is undefined if s is undefined for a variable used in the expression. In the future we omit $\mathbf{i}$ when s suffices to evaluate the expression. We let $[\![P]\!] \in \mathsf{RAM} \setminus \{\top\}$ denote the set of states in which the formula P is satisfied.

In the assertion language, $*$ is the separating conjunction connective, which is interpreted as the operation of separate combination in the corresponding separation domain (by abuse of notation we denote the connective and its interpretation with the same symbol). We also use the iterated version $\circledast$ of the $*$ connective. We can define the usual connectives not mentioned in the syntax definition using the provided ones. The assertion $E \mapsto F$ denotes the set of states in which the heap consists of one cell allocated at the address E and holding the value F, and the stack contains all variables mentioned in E

and F. The assertion $\mathsf{Own}_\pi(\mathtt{x})$ restricts the stack to contain only the variable `x` with the permission π and leaves the heap unconstrained. Variables are treated as resource [62] in the sense that we can separate assertions about variable ownership $\mathsf{Own}_1(\mathtt{x})$ with $*$ in the same way as assertions $E{\mapsto}F$ about ownership of heap cells. For example, variable `x` represented by $\mathsf{Own}_1(\mathtt{x})$ can be split into two permissions $\mathsf{Own}_{1/2}(\mathtt{x})$, each of which permits reading the variable, but not writing to it. Two permissions $\mathsf{Own}_{1/2}(\mathtt{x})$ can later be recombined to obtain the full permission $\mathsf{Own}_1(\mathtt{x})$, which allows both reading from and writing to `x`. We use $\pi_1\mathtt{x}_1, \dots, \pi_k\mathtt{x}_k \Vdash P$ to denote $\mathsf{Own}_{\pi_1}(\mathtt{x}_1) * \dots * \mathsf{Own}_{\pi_k}(\mathtt{x}_k) \wedge P$, abbreviate $1\mathtt{x}$ to `x`, and make the convention that $\Vdash$ binds most loosely and $*$ binds most strongly. The assertion $\mathsf{emp}_\mathbf{s}$ describes the empty stack, and the assertion $\mathsf{emp}_\mathbf{h}$ the empty heap. We denote with emp the assertion $\mathsf{emp}_\mathbf{s} \wedge \mathsf{emp}_\mathbf{h}$. We write _ for a value that is irrelevant and implicitly existentially quantified.

Linked data structures are represented by extending the assertion language with inductive predicate assertions. For example, singly-linked lists are described using the assertion $\mathsf{ls}(E, F)$ denoting the least predicate satisfying

$$\mathsf{ls}(E, F) \Leftrightarrow E \neq F \wedge (E{\mapsto}F \vee \exists X.\, E{\mapsto}X * \mathsf{ls}(X, F)),$$

where X is chosen fresh. Thus, $\mathsf{ls}(E, F)$ represents all of the states in which $E \neq F$ and the heap has the shape of a nonempty acyclic singly-linked list with the first node allocated at the address E and the pointer stored in the last node equal to F. Cyclic lists can be expressed using multiple predicates: e.g., $\mathtt{x} \Vdash \exists X.\, \mathsf{ls}(\mathtt{x}, X) * \mathsf{ls}(X, \mathtt{x})$. Note that $\mathtt{x} \Vdash \mathtt{x}{\mapsto}\mathtt{x}$ is a cycle of length one, while $\mathtt{x} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{x})$ is inconsistent.

Assuming a definition of a C-like structure with a field `F`, in the future we use `x.F` in the assertion language as syntactic sugar for $\mathtt{x} + d$, where d is the offset of the field `F` in the structure. Thus, given the C type

```
struct NODE {
  NODE *Back;
  NODE *Fwd;
  int Data;
};
```

we can describe doubly-linked lists of `NODE`s with the assertion $\mathsf{dll}(E_1, F_1, F_2, E_2)$ denoting the least predicate such that

$$\begin{aligned}\mathsf{dll}(E_1, F_1, F_2, E_2) \Leftrightarrow{} & (F_2 = E_1 \wedge F_1 = E_2 \wedge \mathsf{emp}) \vee \\ & \exists X.\, (E_1.\mathtt{Back}{\mapsto}F_1 * E_1.\mathtt{Fwd}{\mapsto}X * E_1.\mathtt{Data}{\mapsto}\_ * \mathsf{dll}(X, E_1, F_2, E_2))\end{aligned}$$

for a fresh X. The assertion $\mathsf{dll}(E_1, F_1, F_2, E_2)$ represents the states in which the heap has the shape of a (possibly empty) doubly-linked list, where E_1 is the address of the first node of the list, E_2 is the address of the last node, F_1 is the pointer in the `Back` field of the first node, and F_2 is the pointer in the `Fwd` field of the last node. When the list is cyclic, we have $F_2 = E_1$ and $F_1 = E_2$.

Precise and intuitionistic predicates. The following special classes of predicates $p \in \mathcal{P}(\Sigma)$ over separation algebras Σ are important for the future technical development. A predicate p is *precise* [58, 56] if for any state σ there exists at most one substate σ_1 satisfying p: $\sigma = \sigma_1 * \sigma_2$ for some σ_2. If such a substate exists and the $*$ operation is cancellative, then the substate σ_2 is unique and is denoted with $\sigma \backslash p$. By convention, we also say that $\top$ is precise. A predicate p is *intuitionistic* [45] if it is closed under state extension: if p is true of a state σ_1, then for any state σ_2, such that $\sigma_1 * \sigma_2$ is defined, p is also true of $\sigma_1 * \sigma_2$. An assertion is precise or intuitionistic if its denotation is precise or intuitionistic. Consider the following examples of assertions in the language for the domain RAM introduced above.

- $\mathsf{emp_s} \wedge 10{\mapsto}\_$ is precise, but $(\mathsf{emp_s} \wedge 10{\mapsto}\_) \vee \mathsf{emp}$ is not: in a heap where the cell at the address 10 is allocated, both the empty subheap and the subheap containing only the cell satisfy the assertion.

- Neither of the above assertions is intuitionistic, but $(\mathsf{emp_s} \wedge 10{\mapsto}\_) * \mathbf{true}$ is.

- $\mathtt{h} \Vdash \exists X.\, \mathsf{dll}(\mathtt{h}, X, \mathtt{h}, X)$ is imprecise, since it is satisfied both by the empty heap and by a non-empty cyclic doubly-linked list containing the node at the address $\mathtt{h}$. However, $\mathtt{h} \Vdash \exists X, Y.\, \mathtt{h.Back}{\mapsto}X * \mathtt{h.Fwd}{\mapsto}Y * \mathtt{h.Data}{\mapsto}\_ * \mathsf{dll}(Y, \mathtt{h}, \mathtt{h}, X)$ is precise, since it is only satisfied by a non-empty cyclic doubly-linked with a node at the address $\mathtt{h}$.

- $\mathtt{h} \Vdash \mathsf{ls}(\mathtt{h}, \_)$ is imprecise, since it is true of any prefix of a singly-linked list starting from $\mathtt{h}$. However, $\mathtt{h} \Vdash \mathsf{ls}(\mathtt{h}, \mathtt{NULL})$ is precise.

2.1.2 Primitive commands and local functions

The programming languages we consider in this dissertation are parameterised by a set Seq of primitive sequential commands. We define their semantics using forward predicate transformers, which transform predicates over program states represented by elements of a separation algebra Σ. Namely, a forward predicate transformer f_C for a command C maps pre-states to states obtained by executing C from a pre-state. As shown by Calcagno et al. [15], for separation logic to be sound, forward predicate transformers for primitive commands of the programming language have to behave in a local way with respect to the structure present in Σ. The following definition formalises this condition.

Definition 2.3 (Local function). *For a separation algebra $(\Sigma, *)$, a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is local if for any states $\sigma_1, \sigma_2 \in \Sigma$ such that $\sigma_1 * \sigma_2$ is defined, we have*

$$f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}.$$

The *pointwise lifting* of a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ to $\mathcal{P}(\Sigma)^\top$ is a function $f : \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$ defined as follows: for $p \in \mathcal{P}(\Sigma)^\top$

$$f(p) = \begin{cases} \bigsqcup\{f(\sigma) \mid \sigma \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top. \end{cases}$$

Predicate transformers we consider in this dissertation are obtained from functions $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ by pointwise lifting to $\mathcal{P}(\Sigma)^\top$. Note that, if a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is local, then for the corresponding transformer $f : \mathcal{P}(\Sigma)^\top \to \mathcal{P}(\Sigma)^\top$ we have

$$\forall p, q \in \mathcal{P}(\Sigma)^\top.\, f(p * q) \sqsubseteq f(p) * q. \tag{2.1}$$

We say that the predicate transformer is local when it satisfies this property.

Definition 2.3 is a concise way of formulating two conditions that the soundness of separation logic relies on [83]: if $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ is the meaning of a command C, then

(safety monotonicity) if executing C from a state $\sigma_1 * \sigma_2$ results in an error $f(\sigma_1 * \sigma_2) = \top$, then executing C from a smaller state σ_1 also produces an error: $\top \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ implies $f(\sigma_1) = \top$;

(frame property) if executing C from a state σ_1 does not produce an error, then executing C from a larger state $\sigma_1 * \sigma_2$, has the same effect and leaves σ_2 unchanged: in this case we usually have $f(\sigma_1 * \sigma_2) = f(\sigma_1) * \{\sigma_2\}$.

The requirement of locality rules out commands that can check if a cell is allocated in the heap other than by trying to access it and faulting if it is not allocated. For example, consider $D = \mathsf{RAM} = \mathcal{P}(\text{States})^\top$ and the pointwise lifting to $\mathcal{P}(\text{States})^\top$ of the following function $f : \text{States} \to \mathcal{P}(\text{States})^\top$:

$$f(s, h, \mathbf{i}) = \begin{cases} \{(s, h', \mathbf{i})\}, & \text{if } h(10)\downarrow; \\ \{(s, h, \mathbf{i})\}, & \text{otherwise}, \end{cases}$$

where h' is identical to h except it is undefined at 10. The function f defines the forward predicate transformer for a command that disposes of the cell at the address 10 if it is allocated and acts as a no-op if it is not. The function f is not local: take $p = [\![\mathsf{emp}]\!]$ and $q = [\![\mathsf{emp_s} \wedge 10 \mapsto 0]\!]$, then $f(p * q) = f([\![\mathsf{emp_s} \wedge 10 \mapsto 0]\!]) = [\![\mathsf{emp}]\!]$ and

$$f(p) * q = f([\![\mathsf{emp}]\!]) * [\![\mathsf{emp_s} \wedge 10 \mapsto 0]\!] = [\![\mathsf{emp}]\!] * [\![\mathsf{emp_s} \wedge 10 \mapsto 0]\!] = [\![\mathsf{emp_s} \wedge 10 \mapsto 0]\!],$$

hence, the inequality $f(p * q) \sqsubseteq f(p) * q$ does not hold.

As we now illustrate, the predicate transformers corresponding to common heap-manipulating commands are local.

Examples of local functions. Let E, F range over expressions without logical variables and B over Boolean expressions:

$$\begin{array}{rcl}
\mathtt{x} & \in & \mathrm{Vars} \\
E, F & ::= & \mathtt{NULL} \mid \mathtt{x} \mid E + F \mid \ldots \\
B & ::= & E = F \mid E \neq F \mid \ldots
\end{array}$$

The set SeqRAM defined below contains primitive sequential commands often used in heap-manipulating programs:

$$\begin{array}{rcll}
\mathsf{SeqRAM} & ::= & \mathtt{skip} & \text{no-op} \\
& \mid & \mathtt{x} = E & \text{assignment} \\
& \mid & \mathtt{x} = [E] & \text{memory read} \\
& \mid & [E] = F & \text{memory write} \\
& \mid & \mathtt{x} = \mathtt{new} & \text{allocation} \\
& \mid & \mathtt{delete}\ E & \text{deallocation} \\
& \mid & \mathtt{assume}(B) & \text{assume } B \text{ holds}
\end{array}$$

Here square brackets denote pointer dereferencing. The $\mathtt{assume}(B)$ command acts as a filter on the state space of programs—B is assumed to be true after $\mathtt{assume}(B)$ is executed.

We define the forward predicate transformers $f_C : \mathsf{RAM} \to \mathsf{RAM}$ for primitive commands SeqRAM over the separation domain RAM using the transition relation $\rightsquigarrow$: $\mathsf{SeqRAM} \times \mathrm{States} \times (\mathrm{States} \cup \{\top\})$ shown in Figure 2.3. The transformers are pointwise liftings to RAM of functions $f_C : \mathrm{States} \to \mathsf{RAM}$ defined as follows: for $\sigma \in \mathrm{States}$

$$f_C(\sigma) = \bigsqcup \big\{ \{\!|\sigma'|\!\} \mid C, \sigma \rightsquigarrow \sigma' \big\}.$$

It is not difficult to show that for each primitive command $C \in \mathsf{SeqRAM}$ the function f_C is local.

Note that the set States contains program states with permissions for variables less than 1. Such states are not usually encountered during a program's execution, but are used in separation logic to interpret assertions representing parts of complete states. To define a logic with assertions interpreted over the domain RAM and prove it sound, we have to define the behaviour of primitive sequential commands on these states (e.g., as above) and ensure that the commands satisfy the locality condition on them. Once the soundness of the logic is established, the information about permissions can be erased from the model of states used to define program semantics without changing the meaning.

$$
\begin{array}{lll}
\texttt{skip}, (s, h, \mathbf{i}) & \rightsquigarrow & (s, h, \mathbf{i}) \\
\texttt{x} = E, (s[\texttt{x} : (u, 1)], h, \mathbf{i}) & \rightsquigarrow & (s[\texttt{x} : ([\![E]\!]_{s[\texttt{x}:(u,1)]}, 1)], h, \mathbf{i}) \\
\texttt{x} = [E], (s[\texttt{x} : (u, 1)], h[w : \mathbf{Cell}(b)], \mathbf{i}) & \rightsquigarrow & (s[\texttt{x} : (b, 1)], h[w : \mathbf{Cell}(b)], \mathbf{i}), w = [\![E]\!]_{s[\texttt{x}:(u,1)]} \\
[E] = F, (s, h[[\![E]\!]_s : \mathbf{Cell}(u)], \mathbf{i}) & \rightsquigarrow & (s, h[[\![E]\!]_s : \mathbf{Cell}([\![F]\!]_s)], \mathbf{i}) \\
\texttt{x} = \texttt{new}, (s[\texttt{x} : (u, 1)], h, \mathbf{i}) & \rightsquigarrow & (s[\texttt{x} : (b, 1)], h[b : \mathbf{Cell}(w)], \mathbf{i}), \text{ if } h(b)\uparrow \\
\texttt{delete } E, (s, h[[\![E]\!]_s : \mathbf{Cell}(u)], \mathbf{i}) & \rightsquigarrow & (s, h, \mathbf{i}), \text{ if } h([\![E]\!]_s)\uparrow \\
\texttt{assume}(B), (s, h, \mathbf{i}) & \rightsquigarrow & (s, h, \mathbf{i}), \text{ if } [\![B]\!]_s = \mathbf{true} \\
\texttt{assume}(B), (s, h, \mathbf{i}) & \not\rightsquigarrow & \qquad\quad \text{if } [\![B]\!]_s = \mathbf{false} \\
C, (s, h, \mathbf{i}) & \rightsquigarrow & \top, \qquad \text{otherwise}
\end{array}
$$

Figure 2.3: Transition relation for primitive commands SeqRAM. $\top$ indicates that the command faults. $\not\rightsquigarrow$ is used to denote that the command does not fault, but gets stuck. We assume a function $[\![B]\!]_s \in \{\mathbf{true}, \mathbf{false}\}$ that evaluates a Boolean expression B with respect to the stack s.

2.1.3 Sequential separation logic

Abstract formulation. We consider a version of sequential separation logic that is a Hoare logic for a programming language with the following syntax:

$$
\begin{array}{rcll}
C & ::= & \mathsf{Seq} & \text{primitive sequential command} \\
 & | & C; C & \text{sequential composition} \\
 & | & C + C & \text{non-deterministic choice} \\
 & | & C^* & \text{looping}
\end{array}
$$

parameterised with a set Seq of primitive sequential commands. When Seq includes the `assume` command, the standard commands for conditionals and loops can be defined in this language as syntactic sugar:

$$\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 = (\texttt{assume}(B); C_1) + (\texttt{assume}(\neg B); C_2)$$

and

$$\texttt{while } B \texttt{ do } C = (\texttt{assume}(B); C)^*; \texttt{assume}(\neg B)$$

Note that we do not provide a command for declaring variables: permissions for them have to be supplied in the program's initial state.

Abstract separation logic, which we present here, does not prescribe a particular model of states for programs in the above language. Therefore, we assume

- a separation algebra $(\Sigma, *)$ representing program states;
- local functions $f_C : \Sigma \to \mathcal{P}(\Sigma)^\top$ defining the semantics of primitive sequential commands $C \in \mathsf{Seq}$,

out of which we construct

$$\frac{f_C(\llbracket P \rrbracket) \sqsubseteq \llbracket Q \rrbracket}{\{P\}\ C\ \{Q\}}\ \textsc{Prim} \qquad \frac{\{P\}\ C_1\ \{Q\} \quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}\ \textsc{Seq}$$

$$\frac{\{P\}\ C_1\ \{Q\} \quad \{P\}\ C_2\ \{Q\}}{\{P\}\ C_1 + C_2\ \{Q\}}\ \textsc{Choice} \qquad \frac{\{P\}\ C\ \{P\}}{\{P\}\ C^*\ \{P\}}\ \textsc{Loop}$$

$$\frac{\{P_1\}\ C\ \{Q_1\} \quad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}\ \textsc{Disj} \qquad \frac{\{P_1\}\ C\ \{Q_1\} \quad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}\ \textsc{Conj}$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}\ C\ \{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\ C\ \{Q_1\}}\ \textsc{Conseq} \qquad \frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame}$$

Figure 2.4: Proof rules of sequential separation logic

- a concrete separation domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, *, e)$ such that $D = \mathcal{P}(\Sigma)^\top$;
- local forward predicate transformers $f_C : D \to D$.

We assume that the algebra Σ is such that the corresponding domain D has a unit element. Here and in the future, the predicate transformers are obtained from the corresponding functions by pointwise lifting. We further assume an assertion language for denoting elements of the domain D distinct from $\top$, including $\vee$, $\wedge$, $\Rightarrow$, and $*$ connectives with the expected interpretation, and the assertion emp denoting the empty state e. Tautological assertions are those whose meaning is Σ. We denote with $\llbracket P \rrbracket \in D \backslash \{\top\}$ the meaning of the formula P in the domain D. By choosing a particular algebra Σ, functions f_C, and an assertion language, we can obtain concrete instantiations of the abstract logic presented here.

The judgements of sequential separation logic are Hoare triples $\{P\}\ C\ \{Q\}$, and the proof rules are summarised in Figure 2.4. Most of the proof rules are standard rules of Hoare logic. We have a single axiom for primitive commands (Prim), which allows any pre- and postconditions consistent with the predicate transformer for the command. This axiom is usually specialised to several syntactic versions in concrete instances of the logic. The conjunction rule (Conj) is useful for combining the results of two proofs; the disjunction rule (Disj) for doing case splits.

Informally, a judgement $\{P\}\ C\ \{Q\}$ in separation logic means that, if the program C is run from an initial state satisfying P, then it is safe (i.e., it does not dereference any invalid pointers), and the final state (if the program terminates) satisfies Q. Lying behind this interpretation is a semantics of commands which results in a memory fault when accessing memory locations not guaranteed to be allocated, as formalised in the locality condition from Section 2.1.2. Thus, the validity of $\{P\}\ C\ \{Q\}$ ensures that P describes all the memory (except that which gets freshly allocated) that may be accessed during the execution of C—the *footprint* of C. Such interpretation of Hoare triples validates the frame rule of separation logic (Frame), which states that if P ensures C's footprint is allocated, then executing C in the presence of additional memory R results in the same behaviour, and C does not touch the extra memory. Thus, the logic implements the principle of local reasoning [57]:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

It is local reasoning and its technical formulation, the frame rule, that allows giving simple proofs to heap-manipulating programs in separation logic.

Logical variables. We say that a separation algebra Σ is an *algebra with logical variables*, if for some separation algebra Σ' we have $\Sigma = \Sigma' \times \text{Ints}$ and the $*$ operation on Σ is defined as follows:

$$(\sigma_1, \mathbf{i}_1) * (\sigma_2, \mathbf{i}_2) = (\sigma_1 * \sigma_2, \mathbf{i}_1 * \mathbf{i}_2),$$

where the set of interpretations Ints and $*$ on interpretations are defined in Section 2.1.1. Given a function $f : \Sigma' \to \mathcal{P}(\Sigma')^\top$ on the underlying algebra without logical variables, we can lift it to a function $f : \Sigma \to \mathcal{P}(\Sigma)^\top$ on the algebra with logical variables as follows:

$$f(\sigma, \mathbf{i}) = \begin{cases} f(\sigma) \times \{\mathbf{i}\}, & \text{if } f(\sigma) \neq \top; \\ \top, & \text{if } f(\sigma) = \top. \end{cases}$$

When Σ is an algebra with logical variables, we can extend the assertion language with quantifiers:

$$P \ ::= \ \dots \mid \exists X.\, P \mid \forall X.\, P$$

where the satisfaction relation is defined as follows:

$$\begin{array}{lcl} (\sigma, \mathbf{i}) \models \exists X.\, P & \Leftrightarrow & \exists u \in \text{Values}.\, (\sigma, \mathbf{i}[X : u]) \models P \\ (\sigma, \mathbf{i}) \models \forall X.\, P & \Leftrightarrow & \forall u \in \text{Values}.\, (\sigma, \mathbf{i}[X : u]) \models P \end{array}$$

When the functions f_C defining the semantics of primitive sequential commands are lifted from functions on the underlying algebra without logical variables, we can soundly extend separation logic with the following two proof rules for manipulating logical variables:

$$\frac{\{P\}\ C\ \{Q\}}{\{\exists X.\, P\}\ C\ \{\exists X.\, Q\}}\ \textsc{Exists}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{\forall X.\, P\}\ C\ \{\forall X.\, Q\}}\ \textsc{Forall}$$

Note that the algebra States used to construct the domain RAM is an algebra with logical variables and that the corresponding assertion language contains $\exists$ and $\forall$ (the latter defined as $\neg\exists\neg$), whose semantics is consistent with the above definition. The functions f_C for commands $C \in \mathsf{SeqRAM}$ defined in Section 2.1.2 satisfy the condition required for the soundness of Exists and Forall.

$$\frac{}{\{\mathsf{emp}\}\ \mathtt{skip}\ \{\mathsf{emp}\}}\ \textsc{Skip}$$

$$\frac{}{\{\mathtt{x}, O \Vdash X = E \wedge \mathsf{emp_h}\}\ \mathtt{x} = E\ \{\mathtt{x}, O \Vdash \mathtt{x} = X \wedge \mathsf{emp_h}\}}\ \textsc{Assn}$$

$$\frac{}{\{\mathtt{x}, O \Vdash X = E \wedge X \mapsto Y\}\ \mathtt{x} = [E]\ \{\mathtt{x}, O \Vdash \mathtt{x} = Y \wedge X \mapsto Y\}}\ \textsc{Lookup}$$

$$\frac{(O \Vdash E \mapsto \_) \Rightarrow F = F}{\{O \Vdash E \mapsto \_\}\ [E] = F\ \{O \Vdash E \mapsto F\}}\ \textsc{Mutate}$$

$$\frac{}{\{\mathtt{x} \Vdash \mathsf{emp_h}\}\ \mathtt{x} = \mathtt{new}\ \{\mathtt{x} \Vdash \mathtt{x} \mapsto \_\}}\ \textsc{New}$$

$$\frac{}{\{O \Vdash E \mapsto \_\}\ \mathtt{delete}\ E\ \{O \Vdash \mathsf{emp_h}\}}\ \textsc{Delete}$$

$$\frac{}{\{O \Vdash B \wedge \mathsf{emp_h}\}\ \mathtt{assume}(B)\ \{O \Vdash B \wedge \mathsf{emp_h}\}}\ \textsc{Assume-T}$$

$$\frac{}{\{O \Vdash \neg B \wedge \mathsf{emp_h}\}\ \mathtt{assume}(B)\ \{\mathbf{false}\}}\ \textsc{Assume-F}$$

Figure 2.5: Axioms for primitive commands SeqRAM

A logic for the domain RAM. The abstract version of the logic presented above can be specialised to a logic for the domain RAM as follows. Consider the assertion language for RAM presented in Section 2.1.1 and let the set of primitive sequential commands Seq = SeqRAM (Section 2.1.2). In this case the axiom Prim of abstract separation logic can be instantiated to several syntactic versions, for different commands from SeqRAM (Figure 2.5). The axioms are the same as in [66, 62] modulo treating variables as resource in heap-manipulating commands. In the axioms and the following, O ranges over assertions of the form $\pi_1\mathtt{x}_1, \ldots, \pi_k\mathtt{x}_k$, i.e., we use O to supply the permissions for variables necessary for executing the command. We also allow O to be empty, in which case we interpret $O \Vdash P$ as $\mathsf{emp_s} \wedge P$. The following theorem shows that the axioms in Figure 2.5 are valid instances of Prim.

Theorem 2.4. *The axioms for primitive commands* SeqRAM *are sound with respect to the corresponding predicate transformers over the domain* RAM*: for any axiom* $\{P\}\ C\ \{Q\}$ *we have* $f_C(\llbracket P \rrbracket) \sqsubseteq \llbracket Q \rrbracket$.

The proof is an adaptation of the proofs from [45, 62] to our setting.

Note that, in accordance with the principle of local reasoning, the axioms are local in the sense that they mention only the cells that the command actually accesses. In the case when extra memory is present, we can apply their global versions, obtained by closing them under the frame rule. For example, for New we have

$$\frac{}{\{(\mathtt{x} \Vdash \mathsf{emp_h}) * P\}\ \mathtt{x} = \mathtt{new}\ \{(\mathtt{x} \Vdash \mathtt{x} \mapsto \_) * P\}}$$

Additionally, from the rules in Figure 2.4 and axioms Assume-T and Assume-F we can derive proof rules for `if` and `while` commands:

$$\frac{P \Rightarrow B = B \quad \{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \{Q\}}\ \textsc{Cond}$$

$$\frac{P \Rightarrow B = B \quad \{P \wedge B\}\ C\ \{P\}}{\{P\}\ \mathtt{while}\ B\ \mathtt{do}\ C\ \{P \wedge \neg B\}}\ \textsc{While}$$

The premiss $P \Rightarrow B = B$ ensures that the variables mentioned in B are in the stack of the precondition: $P \Rightarrow B = B$ is false in a state for which this is not the case.

2.1.4 Concurrent separation logic

We now consider a variant of concurrent separation logic [56] for a concurrent programming language with static locks and threads, in which programs consist of a parallel composition $C_1 \parallel \ldots \parallel C_n$ of n threads $C_1, \ldots, C_n$ that use m locks $\ell_1, \ldots, \ell_m$ for synchronisation. The code of threads is written in the language of Section 2.1.3 extended with (syntactically scoped) critical regions over the available locks:

$$C \quad ::= \quad \ldots \mid \mathtt{acquire}(\ell_k);\ C;\ \mathtt{release}(\ell_k)$$

We assume that primitive commands of the sequential subset of the language are executed atomically.

Concurrent separation logic achieves modular reasoning about programs in this language by partitioning the program state (variables and the heap) into several disjoint parts: thread-local parts (one for each thread) and protected parts (one for each free lock, i.e., a lock that is not held by any thread). A thread-local part can only be accessed by the corresponding thread. To access the part protected by a lock, a thread has to acquire the lock first. It is important to note that this partitioning is not a part of the program itself, but is enforced by proofs in the logic to enable modular reasoning. For such a partitioning to exist, the program has to satisfy what O'Hearn terms the Ownership Hypothesis [56]: "A code fragment can access only those portions of state that it owns". In other words, the program has to admit a notion of ownership of state parts by threads and locks. The ownership relation is not required to be static, i.e., the logic permits ownership transfer of variables and heap cells between areas owned by different threads and locks.

The benefit of this view of the program state for modular reasoning is that, while reasoning about a given thread, one does not have to consider the local states of other threads, since these local states cannot influence its behaviour. To reason modularly about parts of the state protected by locks, the logic associates with every lock ℓ_k an assertion I_k—its *resource invariant*—that describes how the part of the state protected by the lock looks when the lock is free. For example, when $D = \mathsf{RAM}$, a resource invariant for a lock can state that the lock protects a cyclic doubly-linked list with a sentinel node pointed to by the variable `h`:

$$\mathtt{h} \Vdash \exists X, Y.\, \mathtt{h.Back}\mapsto X * \mathtt{h.Fwd}\mapsto Y * \mathtt{h.Data}\mapsto \_ * \mathsf{dll}(Y, \mathtt{h}, \mathtt{h}, X).$$

For any given thread, resource invariants restrict how other threads can change the protected state, and hence, allow reasoning about the thread in isolation.

The judgements of concurrent separation logic are of the form $I \vdash \{P\}\ C\ \{Q\}$, where C is a command in the code of a thread, P and Q describe the local state of the thread before and after executing C, and I is a vector of resource invariants I_k for the locks used in the program. The proof rules of sequential separation logic are adapted to the new setting by prefixing every triple in them with $I \vdash$.

When a thread acquires a lock, it gets the ownership of its resource invariant. This is formalised in the logic with the following axiom:

$$\frac{}{I \vdash \{\mathsf{emp}\}\ \mathtt{acquire}(\ell_k)\ \{I_k\}}\ \textsc{Acquire}$$

Before releasing the lock, the thread must re-establish the corresponding resource invariant. After the lock is released, the thread gives up the ownership of its resource invariant, as follows from the following axiom:

$$\frac{}{I \vdash \{I_k\}\ \mathtt{release}(\ell_k)\ \{\mathsf{emp}\}}\ \textsc{Release}$$

As usual, we can obtain global versions of the axioms Acquire and Release by closing them under the frame rule:

$$\frac{}{I \vdash \{P\}\ \mathtt{acquire}(\ell_k)\ \{P * I_k\}}$$

$$\frac{}{I \vdash \{P * I_k\}\ \mathtt{release}(\ell_k)\ \{P\}}$$

The original concurrent separation logic also considers nested parallel compositions, handled with the following proof rule:

$$\frac{I \vdash \{P_1\}\ C_1\ \{Q_1\} \quad I \vdash \{P_2\}\ C_2\ \{Q_2\}}{I \vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

We do not consider nested parallel compositions in this dissertation, but in Chapter 6 we show how to handle dynamic thread creation, a more general programming construct than parallel composition.

Brookes [12] gives a denotational semantics for concurrent separation logic based on action traces (later generalised and simplified in [11, 39, 15]), and shows that the logic is sound provided resource invariants I_k are precise and the $*$ operation is cancellative. The famous Reynolds's counterexample [56] demonstrating the need for this restriction involves the conjunction rule (Conj). However, to date it has been an open question whether concurrent separation logic without the conjunction rule is sound when the restriction on resource invariants is dropped: Brookes's whole proof depends on precision of resource invariants. In Chapter 3, we give a positive answer to this question and demonstrate the connections between such a non-standard variant of the logic and a class of program analyses.

We do not use Brookes's semantics in this dissertation; in Section 3.2 we present a simple interleaving-based operational semantics for the concurrent programming language considered here that is sufficient for our purposes.

2.2 Abstract interpretation

In this dissertation, by program analyses we mean analyses based on abstract interpretation [20]. Here we provide the necessary background from abstract interpretation for sequential programs.

As is usual in program analysis literature, we abstract away from a particular syntax of the programming language and represent sequential programs by their control-flow graphs (CFG). A CFG over a set Prim of primitive commands is defined as a tuple $(N, T, \mathsf{start}, \mathsf{end})$, where N is the set of program points, $T \subseteq N \times \mathsf{Prim} \times N$ the control-flow relation, start and end distinguished starting and final program points. Throughout this dissertation we assume, without loss of generality, that control-flow relations have no edges leading to start or going out of end. We note that a command in the language of Section 2.1.3 or the code of a thread in the language of Section 2.1.4 can be translated to a CFG.

Consider a program S in the language of Section 2.1.3 with a CFG $(N, T, \mathsf{start}, \mathsf{end})$ over the set of primitive commands Seq. We assume

- a set Σ of program states;
- functions $f_C : \Sigma \to \mathcal{P}(\Sigma)^\top$ defining the semantics of primitive sequential commands $C \in \mathsf{Seq}$,

out of which we construct

- a domain $(D, \sqsubseteq, \bigsqcup, \bot, \top)$ such that $D = \mathcal{P}(\Sigma)^\top$;
- forward predicate transformers $f_C : D \to D$.

In abstract interpretation literature D is usually referred to as a concrete domain, and the predicate transformers f_C as concrete transfer functions. We define the operational semantics of the program by the transition relation $\rightarrow_S$: $(N \times \Sigma) \times (N \times (\Sigma \cup \{\top\}))$ transforming pairs of program points and states. It is defined as the least relation satisfying the following rules:

$$\frac{(v, C, v') \in T \quad f_C(\{\sigma\}) \sqsubset \top \quad \sigma' \in f_C(\{\sigma\})}{v, \sigma \rightarrow_S v', \sigma'}$$

$$\frac{(v, C, v') \in T \quad f_C(\{\sigma\}) = \top}{v, \sigma \rightarrow_S v', \top}$$

We denote with $\rightarrow_S^*$ the reflexive and transitive closure of $\rightarrow_S$.

Consider $p_0 \in D$ representing the set of initial states of the program S. The *collecting semantics* of the program S is a function $R(p_0) : N \rightarrow D$ that for every program point gives the set of states reachable at this point when the program executes from an initial state in p_0:

$$R(p_0, v) = \bigsqcup \big\{\sigma \mid \mathsf{start}, \sigma_0 \rightarrow_S^* v, \sigma \wedge \{\sigma_0\} \sqsubseteq p_0\big\}.$$

The collecting semantics can alternatively be defined as the least fixed point of the functional $\mathcal{F}(p_0) : (N \rightarrow D) \rightarrow (N \rightarrow D)$, that takes a function G and produces a function $\widetilde{G}$ as follows: $\mathcal{F}(p_0)(G) = \widetilde{G}$, where $\widetilde{G}(\mathsf{start}) = p_0$ and for every program point $v' \in N \backslash \{\mathsf{start}\}$

$$\widetilde{G}(v') = \bigsqcup_{(v,C,v') \in T} f_C(G(v)).$$

That is, to compute reachable states at any program point (except for the initial one) we consider all the edges in the CFG of the program that lead to this point and take the join of the predicate transformers for the commands at these edges with respect to the states at the source program points. The functional $\mathcal{F}(p_0)$ is monotone under the pointwise extension of $\sqsubseteq$, hence, by Tarski's fixed-point theorem it has least fixed point $\mathsf{lfp}(\mathcal{F}(p_0))$. It is then easy to show that $R(p_0) \sqsubseteq \mathsf{lfp}(\mathcal{F}(p_0))$.[1]

Program analyses based on abstract interpretation typically compute over-approximations of the collecting semantics, using its characterisation as a fixed point. One way to construct an abstract interpretation is to devise

- an abstract domain $(D^\sharp, \sqsubseteq, \bigsqcup, \bot, \top)$ representing abstract states of the program;
- a monotone concretisation function $\gamma : D^\sharp \rightarrow D$ giving the meaning of abstract states in the concrete domain D;
- abstract transfer functions $f_C^\sharp : D^\sharp \rightarrow D^\sharp$ defining the abstract semantics of sequential commands $C \in \mathsf{Seq}$

[1]We have an inequality instead of an exact equality here because the operational semantics stops executing the program when it encounters an error $\top$, whereas this is not the case for the characterisation of the collecting semantics defined here.

such that the abstract transfer functions over-approximate the concrete ones:

$$\forall C \in \mathsf{Seq}.\, \forall p \in D^\sharp.\, f_C(\gamma(p)) \sqsubseteq \gamma(f^\sharp_C(p)). \tag{2.2}$$

Note that we use the same symbols for the order, bottom and top elements, and the join operator for both abstract and concrete domains. Elements of the abstract domain concisely represent potentially infinite sets of concrete states, and abstract transfer functions compute approximate effects of program commands. This allows a program analysis to compute an over-approximation of the meaning of a program as described below.

Some frameworks of abstract interpretation also require an abstraction function $\alpha : D \to D^\sharp$ defining an abstract state representing a given set of concrete states such that α and γ form a Galois connection [20]. As the concretisation function alone is sufficient to formulate all our constructions and results, we do not require α here. For simplicity, we also do not consider the use of widening [20] and assume that all the necessary over-approximation of the effect of commands is done by the abstract transfer functions.

Let $p^\sharp_0 \in D^\sharp$ be an element of the abstract domain representing the set of initial states of the program S. Given the ingredients listed above, we can set up a program analysis on S that computes the abstract states reachable from $p^\sharp_0$ in the abstract semantics of S defined by the abstract transfer functions. The analysis is defined by the functional $\mathcal{F}^\sharp(p^\sharp_0) : (N \to D^\sharp) \to (N \to D^\sharp)$ that takes a function $G^\sharp$ and produces a function $\widetilde{G}^\sharp$ as follows: $\mathcal{F}^\sharp(p^\sharp_0)(G^\sharp) = \widetilde{G}^\sharp$, where $\widetilde{G}^\sharp(\mathsf{start}) = p^\sharp_0$ and for every program point $v' \in N \backslash \{\mathsf{start}\}$

$$\widetilde{G}^\sharp(v') = \bigsqcup_{(v,C,v') \in T} f^\sharp_C(G^\sharp(v)).$$

The result of the analysis is a fixed-point of the functional $\mathcal{F}^\sharp(p^\sharp_0)$. In particular, when the abstract transfer functions are continuous, we can compute the least fixed point of $\mathcal{F}^\sharp(p^\sharp_0)$ iteratively as follows:

$$\mathsf{lfp}(\mathcal{F}^\sharp(p^\sharp_0)) = \bigsqcup_{k=0}^{\infty} (\mathcal{F}^\sharp(p^\sharp_0))^k(\lambda v.\, \bot).$$

The abstract domain and transfer functions are usually set up in such a way that the computation defined by this equation converges after a finite number of steps. Let $R^\sharp(p^\sharp_0)$ be a fixed point of $\mathcal{F}^\sharp(p^\sharp_0)$. Using the condition (2.2), it is easy to show that $R^\sharp(p^\sharp_0)$ over-approximates the collecting semantics:

$$R(\gamma(p^\sharp_0)) \sqsubseteq \gamma(R^\sharp(p^\sharp_0)),$$

where γ and $\sqsubseteq$ are lifted to $N \to D^\sharp$ pointwise. This establishes the soundness of the analysis with respect to the concrete semantics.

Using this soundness statement, we can check safety properties of the program S as follows. Let $\mathsf{safe}^\sharp \in D^\sharp$ be an element of the abstract domain, whose concretisation $\gamma(\mathsf{safe}^\sharp)$

represents the set of safe states of the program S. According to the above soundness statement, to check that the program may not reach an unsafe state, it is sufficient to check that $\forall v \in N.\, (R^\sharp(p_0^\sharp))(v) \sqsubseteq \mathsf{safe}^\sharp$. In particular, in this way we can check invariance properties. A wider class of safety properties can be reduced to this by instrumenting the program with a monitor [78]. Additionally, the information about reachable states computed by the analysis can also be used to enable compiler optimisations.

2.2.1 Examples of abstract domains

In this dissertation, we are mostly interested in heap analyses, which determine the structure of the heap during the execution of a program. These analyses typically do not maintain information about the data stored in heap-allocated data structures, tracking only the shapes the data structures take; for this reason, they are often called shape analyses. The crucial difference between heap analyses and more shallow analyses for heap-manipulating programs, such as pointer analyses, is that the former aim to be precise in the presence of deep heap update—alteration of linked data structures after traversing them for an arbitrary distance. Heap analyses can be used to verify a number of safety properties (including memory safety, data-structure integrity, the absence of memory leaks, and the absence of assertion violations) and have been successfully used as a key ingredient in methods for verifying liveness properties (e.g., [4, 8, 35]).

Many heap analyses have been proposed in recent years. In this section we do not aim to give a comprehensive survey; rather, we briefly describe the abstract domains for heap analysis we refer to in the following chapters. We have adapted the original presentations of the domains to use variables as resource, i.e., to use concrete domains similar to the domain RAM in Section 2.1.1.

IntDom: interval domain [19]. Before describing abstract domains for heap analysis, we give an example of a very simple domain for determining numerical properties of programs without dynamically allocated memory. Consider the programming language of Section 2.1.3 with $\mathsf{Seq} = \{\texttt{skip}, \texttt{x} = E, \texttt{assume}(B)\}$. The interval domain is used to determine ranges of values that variables can take during the execution of a program. We define the set Σ of concrete states as follows:

$$\begin{aligned}
&\text{Values} = \{\ldots, -1, 0, 1, \ldots\}\\
&\text{Vars} = \{\texttt{x}, \texttt{y}, \ldots\}\\
&\Sigma = \text{Vars} \rightharpoonup_{\text{fin}} \text{Values}
\end{aligned}$$

Let the concrete domain $D = \mathcal{P}(\Sigma)^\top$. The concrete transfer functions $f_C : D \to D$ are defined in the same way as the functions over the domain RAM (Section 2.1.2). Let Intervals be the set of intervals of the form $[a, b]$, where $a \in \text{Values} \cup \{-\infty\}$, $b \in \text{Values} \cup \{+\infty\}$, and $a \leq b$. The interval abstract domain is defined as $\mathsf{IntDom} = (\text{Vars} \rightharpoonup_{\text{fin}}$

Intervals) $\cup \{\top\}$, i.e., a non-error element of the domain determines a range of possible values for some of the program variables. The order on the domain is as follows:

$$\begin{aligned}\forall p_1, p_2 \in \mathsf{IntDom}\backslash\{\top\}.\, p_1 \sqsubseteq p_2 \Leftrightarrow (&\mathsf{dom}(p_1) = \mathsf{dom}(p_2) \wedge \\ &\forall \mathtt{x} \in \text{Vars}.\ p_1(\mathtt{x}) = [a_1, b_1] \wedge p_2(\mathtt{x}) = [a_2, b_2] \Rightarrow a_2 \leq a_1 \wedge b_1 \leq b_2)\end{aligned}$$

and $\forall p \in \mathsf{IntDom}.\, p \sqsubseteq \top$. Finally, we define the concretisation function $\gamma : \mathsf{IntDom} \to D$. For $p \in \mathsf{IntDom}\backslash\{\top\}$ let

$$\gamma(p) = \{s \mid \mathsf{dom}(s) = \mathsf{dom}(p) \wedge \forall \mathtt{x} \in \text{Vars}.\, p(\mathtt{x}) = [a, b] \Rightarrow a \leq s(\mathtt{x}) \leq b\}$$

and let $\gamma(\top) = \top$. See [19] for the definition of abstract transfer functions for commonly used expressions E and B. Since our concrete domain treats variables as resource, concrete and abstract transfer functions on the domain have to return $\top$ on a state that does not have permissions for all the variables accessed by the command.

SLL: a domain for singly-linked lists [24]. This is a domain for heap analysis that represents the heap structure using formulae of separation logic's assertion language. In the following chapters we use the domain to give example instantiations of our analyses, and thus present it here in detail.

Consider the programming language of Section 2.1.3 with $D = \mathsf{RAM}$ (Section 2.1.1), $\mathsf{Seq} = \mathsf{SeqRAM}$ (Section 2.1.2), and the concrete transfer functions defined in Section 2.1.2, where we restrict expressions and Boolean expressions as follows:

$$\begin{array}{rcl} \mathtt{x} & \in & \text{Vars} \\ E, F & ::= & \mathtt{NULL} \mid \mathtt{x} \\ B & ::= & E = F \mid E \neq F \end{array}$$

Domains based on separation logic are typically specialised for a particular class of data structures the program manipulates. Technically, this specialisation is done by selecting a subset of the assertion language with inductive predicates used to describe the desired class of data structures.

The analysis we describe here represents sets of concrete program states with sets of symbolic heaps P given by the following grammar:

$$\begin{array}{rcl} \mathtt{x} & \in & \text{Vars} \\ X & \in & \text{LVars} \\ E, F & ::= & \mathtt{NULL} \mid \mathtt{x} \mid X \\ O & ::= & \vec{\mathtt{x}} \\ \mathsf{P} & ::= & \textbf{true} \mid \mathsf{P} \wedge \mathsf{P} \mid E = E \\ \mathsf{S} & ::= & \mathsf{emp_h} \mid \mathsf{S} * \mathsf{S} \mid E \mapsto E \mid \mathsf{ls}(E, E) \mid \textbf{true} \\ Q & ::= & \mathsf{P} \wedge \mathsf{S} \\ P & ::= & O \Vdash \exists \vec{X}.\, Q \end{array}$$

Symbolic heap formulae contain a variable ownership assertion O (Section 2.1.1) with total permissions for all variables in it, a Boolean formula P built from $=$ and $\wedge$, which is insensitive to the heap, and a spatial formula S that expresses heap shape. Formulae are considered up to symmetry of $=$, permutations across $\wedge$ and $*$ (e.g., $\mathsf{P} \wedge B_0 \wedge B_1$ and $\mathsf{P} \wedge B_1 \wedge B_0$ are equated), permutations of variables in the lists O and $\vec{X}$, renaming existentially quantified variables, unit laws for **true** and $\mathsf{emp_h}$, idempotency of $\cdot * \mathbf{true}$ (e.g., $\mathbf{true} * \mathbf{true}$ and **true** are equated), adding or removing consequences of equalities present in the pure part, and interchanging equal (due to the equalities in the pure part) variables in the spatial part. So, $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{v} \Vdash \mathtt{x} = \mathtt{y} \wedge \mathtt{y} = \mathtt{z} \wedge \mathsf{ls}(\mathtt{v}, \mathtt{x})$ and $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{v} \Vdash \mathtt{x} = \mathtt{y} \wedge \mathtt{y} = \mathtt{z} \wedge \mathtt{x} = \mathtt{z} \wedge \mathsf{ls}(\mathtt{v}, \mathtt{y})$ are considered equal. We denote the set of symbolic heaps with SH and define the abstract domain as follows: $\mathsf{SLL} = \mathcal{P}(\mathsf{SH})^{\top}$.

The concretisation function is defined using the interpretation of assertions in Section 2.1.1: for $P \in \mathsf{SH}$ we let $\gamma(P) = [\![P]\!]$ and lift it to SLL pointwise. Recall that $\mathsf{ls}(E, F)$ describes non-empty acyclic singly-linked lists. The assertion **true** in the spatial part of a symbolic heap signals a possible memory leak.

Abstract transfer functions are defined by the analysis in terms of the symbolic execution relation $\leadsto$, the rearrangement relation $\hookrightarrow_E$, and the abstraction relation $\hookrightarrow$. The definition of these relations poses several types of questions about symbolic heaps: entailment of an equality

$$(O \Vdash Q) \vdash (O \Vdash E = F),$$

or of a disequality

$$(O \Vdash Q) \vdash (O \Vdash E \neq F),$$

inconsistency

$$(O \Vdash Q) \vdash \mathbf{false},$$

or testing if some location is guaranteed to be allocated

$$(O \Vdash Q) \vdash (O \Vdash E \mapsto \_ * \mathbf{true}).$$

We also sometimes ask the negations of these questions. Decision procedures for these queries are defined in [24]. For the domain SLL they are simple and almost syntactic checks that do not require calling a sophisticated theorem prover. Before applying any of the relations to a symbolic heap $O \Vdash \exists \vec{X}. Q$, the transfer functions check that the necessary permissions for variables accessed by the command are in O.

The symbolic execution relation $\leadsto$ (Figure 2.6) captures the effect of executing a primitive command from a symbolic heap. Each individual concrete state can be expressed exactly by a symbolic heap, i.e., there is a subset of symbolic heaps which are simply different syntax for concrete states. In the usual concrete semantics, each command only accesses a small portion of the state that forms its footprint. From this perspective, symbolic execution expresses the usual concrete semantics of commands in terms of symbolic

$$
\begin{array}{rcl}
O \Vdash \exists \vec{X}.\, Q & \overset{\texttt{skip}}{\leadsto} & O \Vdash \exists \vec{X}.\, Q \\
O \Vdash \exists \vec{X}.\, Q & \overset{\texttt{x}=E}{\leadsto} & O \Vdash \exists \vec{X}, Y.\, \texttt{x} = E[Y/\texttt{x}] \wedge Q[Y/\texttt{x}] \\
O \Vdash \exists \vec{X}.\, Q & \overset{\texttt{x=new}}{\leadsto} & O \Vdash \exists \vec{X}, Y, Z.\, Q[Y/\texttt{x}] * \texttt{x} \mapsto Z \\
O \Vdash \exists \vec{X}.\, Q & \overset{\texttt{assume}(E=F)}{\leadsto} & O \Vdash \exists \vec{X}.\, Q \wedge E = F,\ \text{if } (O \Vdash Q) \nvdash (O \Vdash E \neq F) \\
O \Vdash \exists \vec{X}.\, Q & \overset{\texttt{assume}(E\neq F)}{\leadsto} & O \Vdash \exists \vec{X}.\, Q,\ \text{if } (O \Vdash Q) \nvdash (O \Vdash E = F) \\
 & & \text{and } (O \Vdash Q) \nvdash \textbf{false} \\
O \Vdash \exists \vec{X}.\, Q * E \mapsto F & \overset{\texttt{delete}\ E}{\leadsto} & O \Vdash \exists \vec{X}.\, Q \\
O \Vdash \exists \vec{X}.\, Q * E \mapsto F & \overset{\texttt{x}=[E]}{\leadsto} & O \Vdash \exists \vec{X}, Y.\, \texttt{x} = F[Y/\texttt{x}] \wedge (Q * E \mapsto F)[Y/\texttt{x}] \\
O \Vdash \exists \vec{X}.\, Q * E \mapsto F_0 & \overset{[E]=F}{\leadsto} & O \Vdash \exists \vec{X}.\, Q * E \mapsto F
\end{array}
$$

Figure 2.6: Symbolic Execution $\leadsto$. Here Y and Z are fresh in the corresponding symbolic heaps.

$$
\begin{array}{rcl}
O \Vdash \exists \vec{X}.\, Q * F \mapsto F_0 & \hookrightarrow_E & O \Vdash \exists \vec{X}.\, Q * E \mapsto F_0,\ \text{if } (O \Vdash Q) \vdash (O \Vdash E = F) \\
O \Vdash \exists \vec{X}.\, Q * \mathsf{ls}(F, F_0) & \hookrightarrow_E & O \Vdash \exists \vec{X}.\, Q * E \mapsto F_0,\ \text{if } (O \Vdash Q) \vdash (O \Vdash E = F) \\
O \Vdash \exists \vec{X}.\, Q * \mathsf{ls}(F, F_0) & \hookrightarrow_E & O \Vdash \exists \vec{X}, Y.\, Q * E \mapsto Y * \mathsf{ls}(Y, F_0), \\
 & & \text{if } (O \Vdash Q) \vdash (O \Vdash E = F) \text{ and } Y \text{ is fresh} \\
O \Vdash \exists \vec{X}.\, Q & \hookrightarrow_E & \top,\ \text{if } (O \Vdash Q) \nvdash (O \Vdash E \mapsto \_ * \textbf{true})
\end{array}
$$

Figure 2.7: Rearrangement $\hookrightarrow_E$

heaps, where the footprint of the command is expressed as one of the formulae that is alternate syntax for a concrete state.

Symbolic execution does not operate on arbitrary pre-states. For instance, its definition for $\texttt{x} = [E]$ requires that the source heap cell be explicitly known. In order to put symbolic heaps into the form required for symbolic execution of a command, we use the rearrangement relation $\hookrightarrow_E$ (Figure 2.7) that transforms an arbitrary symbolic heap, via case analysis, into a set of symbolic heaps where the footprint of the next command is concrete. When rearrangement fails to reveal the required location E, it indicates a potential memory safety violation and returns $\top$.

Abstraction $\hookrightarrow$ (Figure 2.8) then takes the symbolic heaps resulting from symbolic execution and maps them into a finite subdomain of symbolic heaps, ensuring that fixed-point computations converge. Abstraction is accomplished by certain separation logic implications that rewrite a symbolic heap to a logically weaker one. We call a symbolic heap P canonical if it is maximally abstracted, i.e., $P \not\hookrightarrow$. A canonicalisation function can is defined in [24]. This function is based on a fixed sequence of abstraction axiom applications, and transforms a symbolic heap to a canonical symbolic heap abstracting it, i.e., $P \hookrightarrow^* \mathsf{can}(P)$ and $\mathsf{can}(P) \not\hookrightarrow$. A key property of the abstract domain SLL is that the set of consistent and canonical symbolic heaps $\{P \mid P \nvdash \textbf{false} \wedge P \not\hookrightarrow\}$ over a finite

$$
\begin{aligned}
O \Vdash \exists \vec{X}, Y.\, Q &\hookrightarrow O \Vdash \exists \vec{X}.\, Q \\
O \Vdash \exists \vec{X}, Y.\, Q \wedge Y = E &\hookrightarrow O \Vdash \exists \vec{X}.\, Q[E/Y] \\
O \Vdash \exists \vec{X}, Y.\, Q * H_0(E,Y) * H_1(Y,F) &\hookrightarrow O \Vdash \exists \vec{X}.\, Q * \mathsf{ls}(E, \mathtt{NULL}), \\
& \text{ if } (O \Vdash Q) \vdash (O \Vdash F = \mathtt{NULL}) \\
O \Vdash \exists \vec{X}, Y.\, Q * H_0(E,Y) * H_1(Y,F_0) * H_2(F_1,F_2) &\hookrightarrow O \Vdash \exists \vec{X}.\, Q * \mathsf{ls}(E, F_0) * H_2(F_1,F_2), \\
& \text{ if } (O \Vdash Q) \vdash (O \Vdash F_0 = F_1) \\
O \Vdash \exists \vec{X}, Y.\, Q * H(Y,E) &\hookrightarrow O \Vdash \exists \vec{X}.\, Q * \mathbf{true} \\
O \Vdash \exists \vec{X}, Y, Z.\, Q * H_0(Y,Z) * H_1(Z,Y) &\hookrightarrow O \Vdash \exists \vec{X}.\, Q * \mathbf{true}
\end{aligned}
$$

Figure 2.8: Abstraction $\hookrightarrow$. Here $H(E,F)$ stands for either $E \mapsto F$ or $\mathsf{ls}(E,F)$, and Y and Z do not occur other than where explicitly indicated.

number of unquantified variables is finite.

For a command C (respectively, an ownership assertion O), let $\mathsf{vars}(C)$ (respectively, $\mathsf{vars}(O)$) be the set of program variables occurring in C (respectively, O). We define abstract transfer functions as pointwise liftings of the following functions $f^\sharp_C : \mathsf{SH} \to \mathsf{SLL}$. For $C \in \{\mathtt{skip}, \mathtt{x} = E, \mathtt{x} = \mathtt{new}, \mathtt{assume}(E = F), \mathtt{assume}(E \neq F)\}$ we let

$$f^\sharp_C(P_0) = \begin{cases} \top, & \text{if } \mathsf{vars}(C) \not\subseteq \mathsf{vars}(O); \\ \{\mathsf{can}(P_1) \mid P_0 \overset{C}{\rightsquigarrow} P_1\}, & \text{otherwise;} \end{cases}$$

for $C \in \{\mathtt{x} = [E], [E] = F, \mathtt{delete}\ E\}$ we let

$$f^\sharp_C(P_0) = \begin{cases} \top, & \text{if } \mathsf{vars}(C) \not\subseteq \mathsf{vars}(O) \text{ or } P_0 \hookrightarrow_E \top; \\ \{\mathsf{can}(P_2) \mid \exists P_1.\, P_0 \hookrightarrow_E P_1 \wedge P_1 \overset{C}{\rightsquigarrow} P_2\}, & \text{otherwise.} \end{cases}$$

For example, suppose we want to compute the value of the transfer function for the command $\mathtt{x} = [\mathtt{x}]$ on the symbolic heap $\mathtt{x} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL})$. The rearrangement phase will transform the heap into two symbolic heaps $\mathtt{x} \Vdash \mathtt{x} \mapsto \mathtt{NULL}$ and $\mathtt{x} \Vdash \exists X.\, \mathtt{x} \mapsto X * \mathsf{ls}(X, \mathtt{NULL})$ thereby making the information that $\mathtt{x}$ is allocated in the heap explicit. The symbolic execution phase will then symbolically simulate the effect of the command on the heaps producing $\mathtt{x} \Vdash \mathtt{x} = \mathtt{NULL} \wedge \mathsf{emp}_\mathbf{h}$ and $\mathtt{x} \Vdash \exists X, Y.\, \mathtt{x} = X \wedge Y \mapsto X * \mathsf{ls}(X, \mathtt{NULL})$. Finally, the abstraction phase will leave the first heap unchanged and will canonicalise the second heap to $\mathtt{x} \Vdash \mathbf{true} * \mathsf{ls}(\mathtt{x}, \mathtt{NULL})$. Hence, the value of the transfer function is $\{(\mathtt{x} \Vdash \mathtt{x} = \mathtt{NULL} \wedge \mathsf{emp}_\mathbf{h}), (\mathtt{x} \Vdash \mathbf{true} * \mathsf{ls}(\mathtt{x}, \mathtt{NULL}))\}$.

The soundness of abstract transfer functions is justified as follows. Symbolic execution follows the global versions of axioms in Figure 2.5 (i.e., the axioms closed under the frame rule), and rearrangement and abstraction use valid separation logic implications. Thus, it is not difficult to show that for any symbolic heap P if $f^\sharp_C(P) \sqsubset \top$, then the triple $\{P\}\ C\ \{\bigvee\{P' \mid P' \in f^\sharp_C(P)\}\}$ is derivable in sequential separation logic for the domain RAM. It is then possible to prove (2.2) by structural induction on the derivation using Theorem 2.4 and locality of concrete transfer functions f_C.

CDS: a domain for composite data structures [2]. This separation logic-based domain handles composite list-based data structures typically found in device drivers and other systems-level programs. Examples of such data structures include "cyclic doubly-linked lists of acyclic singly-linked lists" and "singly-linked lists of cyclic doubly-linked lists with back-pointers to head nodes". The programming language and the concrete domain are the same as for SLL. The analysis uses symbolic heaps P specified by the following grammar:

$$\begin{array}{rcl}
\mathtt{x} & \in & \text{Vars} \\
X & \in & \text{LVars} \\
E, F & ::= & \mathtt{NULL} \mid \mathtt{x} \mid X \\
O & ::= & \vec{\mathtt{x}} \\
\mathsf{P} & ::= & \mathbf{true} \mid \mathsf{P} \wedge \mathsf{P} \mid E = E \mid E \neq E \\
\mathsf{S} & ::= & \mathsf{emp_h} \mid \mathsf{S} * \mathsf{S} \mid E {\mapsto} E \mid \mathsf{ls}\ \Lambda\ (E, E, E, E) \mid \mathbf{true} \\
Q & ::= & \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S} \\
\Lambda & ::= & \lambda[X, X, X].\, Q \\
P & ::= & O \Vdash Q
\end{array}$$

We write $\Lambda[E_1, E_2, E_3]$ for the symbolic heap obtained by instantiating the parameters of Λ: $(\lambda[X_1, X_2, X_3].\, Q)[E_1, E_2, E_3] = Q[E_1/X_1, E_2/X_2, E_3/X_3]$.

The predicate $\mathsf{ls}\ \Lambda\ (E_1, F_1, F_2, E_2)$ represents a segment of a generic doubly-linked list, where the shape of each node in the list is described by the first parameter Λ, and some links between this segment and the rest of the heap are specified by the other parameters. Parameters E_1 and E_2 denote the (externally visible) memory locations of the first and the last nodes of the list segment. The analysis maintains the links from the outside to these exposed cells, so that the links can be used, say, to traverse the segment. Usually, E_1 denotes the address of the "root" of a data structure representing the first node, such as the head of a singly-linked list. The common use of E_2 is similar. Parameters F_1 and F_2 represent links from the first and last nodes of the list segment to the outside, which the analysis decides to maintain.

The formal definition of ls is similar to the definition of dll in Section 2.1.1. For a parameterised symbolic heap Λ, $\mathsf{ls}\ \Lambda\ (E_1, F_1, F_2, E_2)$ is the least predicate such that

$$\mathsf{ls}\ \Lambda\ (E_1, F_1, F_2, E_2) \Leftrightarrow (F_2 = E_1 \wedge F_1 = E_2 \wedge \mathsf{emp_h}) \vee \\ (\exists X.\, (\Lambda[E_1, F_1, X]) * \mathsf{ls}\ \Lambda\ (X, E_1, F_2, E_2)),$$

where X is chosen fresh. A list segment is empty, or it consists of a node described by an instantiation of Λ and a tail satisfying $\mathsf{ls}\ \Lambda\ (X, E_1, F_2, E_2)$. The generic list predicate can express a variety of data structures:

- When $\Lambda_{\mathbf{s}}$ is $\lambda[X,Y,Z].\,(X.\mathtt{Next}\mapsto Z)$, the formula $\exists Y,U.\,\mathsf{ls}\ \Lambda_{\mathbf{s}}\ (X,Y,Z,U)$ describes a (possibly empty and possibly cyclic) singly-linked list segment from X to Z.
- A doubly-linked list segment $\mathsf{dll}(X,Y,Z,U)$ is expressed by $\mathsf{ls}\ \Lambda_{\mathbf{d}}\ (X,Y,Z,U)$, where $\Lambda_{\mathbf{d}}$ is $\lambda[X,Y,Z].\,(X.\mathtt{Back}\mapsto Y * X.\mathtt{Fwd}\mapsto Z * X.\mathtt{Data}\mapsto\_)$.
- Finally, if Λ is

  $$\lambda[X,Y,Z].\,\exists U,V.\,(X.\mathtt{Next}\mapsto Z * X.\mathtt{Back}\mapsto U * X.\mathtt{Fwd}\mapsto V * X.\mathtt{Data}\mapsto\_ * \\ \mathsf{ls}\ \Lambda_{\mathbf{d}}\ (V,X,X,U)),$$

  then $\exists Y,Z.\,\mathsf{ls}\ \Lambda\ (X,Y,\mathtt{NULL},Z)$ describes a singly-linked list of cyclic doubly-linked lists, where each singly-linked list node is the sentinel node of the cyclic doubly-linked list.

The abstract transfer functions for the domain are defined in [2]. The analysis presented there synthesises new parameterised spatial predicates from old predicates using information found in the abstract states visited during its execution. The new predicates can be defined using instances of the generic list predicate in combination with previously synthesised predicates, thus allowing the abstract domain to handle a variety of complex data structures. We note for the future that the theorem prover for entailments $P_1 \vdash P_2$ between symbolic heaps used by the transfer functions is incomplete for this domain (whereas it is complete for SLL).

Complex abstract domains based on separation logic, such as CDS, often use non-standard widening operators [21] that remove redundant separation logic formulae from abstract states using a theorem prover. Hence, these domains do not always fit into the simple abstract interpretation framework used in this dissertation. Our constructions and proofs can be easily adjusted to accommodate the widening operators.

DLL: a domain for doubly-linked lists. We can specialise the CDS domain to programs manipulating doubly-linked lists by restricting the spatial part of symbolic heaps as follows:

$$\mathsf{S} \ ::= \ \mathsf{emp_h} \mid \mathsf{S} * \mathsf{S} \mid E\mapsto E \mid \mathsf{dll}(E,E,E,E) \mid \mathbf{true}$$

The dll predicate denotes possibly empty doubly-linked lists (Section 2.1.1) and can be represented using the generic list predicate as described above.

Domains based on shape graphs. Some abstract domains for heap analysis represent the shapes of data structures in memory using graph-like structures called shape graphs. Like domains based on separation logic, these domains are typically specialised for a particular class of data structures. Examples include a domain by Manevich et al. for singly-linked lists [49] and a domain by Lev-Ami et al. for a wider range of data structures, including singly- and doubly-linked lists, and binary trees [48].

TVLA is a parametric framework for heap analysis based on abstract interpretation [72]. Its abstract domain represents the structure of the heap using 3-valued logical structures. The framework can be instantiated in different ways by varying the predicates used in the 3-valued logic.

Chapter 3

Static locks

The straightforward way to analyse a concurrent heap-manipulating program automatically is to enumerate the interleavings of executions of its threads [80]. Unfortunately, this approach leads to state-space explosion and unscalability. Our goal is to create a heap analysis for concurrent programs that is scalable, sound, and accurate. We do so by constructing a heap analysis that avoids enumerating interleavings. In this chapter, we propose a novel framework for constructing thread-modular program analyses, which is particularly suitable for heap analyses due to the locality exhibited by the semantics of heap manipulation (Section 3.3). Our framework is parametric in the sequential heap analysis domain and can be instantiated with any abstract separation domain, i.e., an abstract domain with a separating conjunction-like operation defined on it. We give several examples of abstract separation domains (Section 3.3.1) and present an instantiation of the framework with a sequential heap analysis based on separation logic (Section 3.3.2).

In this chapter, we demonstrate the main ideas behind the analysis on a simple concurrent programming language with static locks and threads introduced in Section 2.1.4. The subsequent chapters deal with more advanced language features.

Our analysis is inspired by concurrent separation logic: it infers a resource invariant associated with each lock that describes the part of the heap protected by the lock and has to be preserved by every thread. For any given thread, the resource invariant restricts how other threads can interfere with it. Thus, if resource invariants are known, analysing a concurrent program does not require enumerating interleavings and can be done using a sequential heap analysis. The challenge is to infer the resource invariants.

A resource invariant describes two orthogonal kinds of information: it simultaneously carves out the part of the heap protected by the lock and defines the possible shapes that this part can have during program execution. We show that, if we specify the borders of the part of the heap protected by a lock (i.e., the former kind of information), then we can compute the shape of the part (i.e., the latter kind of information) by repeatedly performing heap analysis on each individual thread, but not on the whole program—performing the analysis thread-modularly. The analysis is able to establish that the program being analysed is memory safe, does not leak memory, and does not have data races.

We specify the borders of the part of the heap protected by a lock with *entry points*—program variables such that the part of the heap protected by the lock can be defined as everything that is reachable from them. Fortunately, we find that entry points can be inferred by existing automatic tools [63, 73, 17]. Moreover, the soundness of our analysis does not depend on the particular association of locks and entry points: the analysis can be used with any association.

We show that our analysis can be viewed as generating proofs in a variant of concurrent separation logic without the conjunction rule and the restriction that the resource invariants be precise (Section 3.5). We also prove the soundness of the analysis and both the standard and our variants of the logic in a uniform framework, thereby resolving the open question about the soundness of concurrent separation logic without the conjunction rule and the precision restriction (Section 3.4).

3.1 Thread-modular heap analysis by example

Consider the programming language of Section 2.1.4 in the case when $\mathsf{Seq} = \mathsf{SeqRAM}$ (Section 2.1.2). In our examples throughout the dissertation, we extend programming languages with additional C-like syntax, in particular, C structures. We assume that each field in a structure takes one memory cell. We also use generalisations of `new` and `delete` that allocate and deallocate several memory cells at once.

The example program in Figure 3.1 represents a typical pattern occurring in systems code, such as Windows device drivers. In this case two concurrently executing threads are accessing the same cyclic doubly-linked list protected by a lock ℓ. The list is accessed via a sentinel head node pointed to by a variable `h`, which is also protected by the lock. In this example `thread1` adds nodes to the head of the list and `thread2` removes nodes from the head of the list. The procedure `initialise` can be used to initialise the list before spawning the threads.

When applied to this code, our analysis establishes that the area of the heap protected by the lock—its resource invariant—has the shape of a cyclic doubly-linked list and that the program is memory safe (i.e., it does not dereference invalid pointers), does not leak memory, and has no data races (including races on heap cells).

The analysis uses the domain DLL of Section 2.2.1 as the domain of the underlying sequential heap analysis. It first calls a tool for analysing correlations between locks and program variables, such as [63, 73, 17], to determine that the variable `h` is protected by the lock ℓ. The variable `h` becomes an entry point associated with the lock ℓ: the part of the heap protected by the lock is reachable from the entry point. The analysis is performed iteratively. On each iteration, we analyse the code of each thread and discover symbolic

```
struct NODE {
  NODE *Back;
  NODE *Fwd;
  int Data;
};
LOCK ℓ;
NODE *h;

thread1() {
  int data;
  NODE *n;

  while (nondet()) {
    ...
    acquire(ℓ);
    n = new NODE;
    n->Data = data;
    n->Fwd = h->Fwd;
    n->Back = h;
    h->Fwd = n;
    n->Fwd->Back = n;
    release(ℓ);
    ...
  }
}
```

```
thread2() {
  int data;
  NODE *n;

  while (nondet()) {
    ...
    acquire(ℓ);
    n = h->Fwd;
    if (n != h) {
      n->Back->Fwd = n->Fwd;
      n->Fwd->Back = n->Back;
      data = n->Data;
      delete n;
    }
    release(ℓ);
    ...
  }
}

initialise() {
  h = new NODE;
  h->Back = h;
  h->Fwd = h;
}
```

Figure 3.1: Example program. `nondet()` represents non-deterministic choice.

heaps describing new shapes the part of the heap protected by each lock can take—new disjuncts in its resource invariant. On the next iteration each thread is re-analysed taking the newly discovered disjuncts into account. This loop is performed until no new disjuncts in the resource invariant are discovered, i.e., until we reach a fixed point on the value of the resource invariant. Note that the particular order of the iteration is not important for the soundness of the analysis. In the example below we chose an order that is convenient to illustrate how the analysis works.

As a first step, we run the underlying sequential heap analysis on the `initialise` function to determine the initial approximation

$$I^0 = \mathtt{h} \Vdash \mathtt{h.Back}\mapsto\mathtt{h} * \mathtt{h.Fwd}\mapsto\mathtt{h} * \mathtt{h.Data}\mapsto\_$$

of the resource invariant associated with the lock ℓ. The initial states of the threads in this case are `emp`.

First iteration. We run the underlying sequential heap analysis on the code of `thread1` with the treatment for `acquire` and `release` commands described below. The analysis performs a fixed-point computation to determine the reachable states at all program points in `thread1`. Suppose the analysis reaches line 14 with an abstract state p. Upon acquiring the lock ℓ the thread gets ownership of the part of the heap protected by the lock. We mirror this in the analysis by $*$-conjoining the current approximation I^0 of the resource invariant associated with the lock ℓ to the current state p yielding $p * I^0$. The analysis of the code in lines 15–20 starting from this state then gives us the state

$$p_1 = p * (\mathtt{h}, \mathtt{n} \Vdash \mathtt{h.Back} \mapsto \mathtt{n} * \mathtt{h.Fwd} \mapsto \mathtt{n} * \mathtt{h.Data} \mapsto \_ * \mathtt{n.Back} \mapsto \mathtt{h} * \mathtt{n.Fwd} \mapsto \mathtt{h} * \mathtt{n.Data} \mapsto \_)$$

at line 21. Upon releasing the lock ℓ the thread has to give up the ownership of the part of the heap protected by the lock. This means that the analysis has to split the current heap p_1 into two parts, one of which becomes the local heap of the thread (the part of the heap that the thread owns) and the other is added as a new disjunct to the resource invariant. We compute the splitting in the following way: the part of the heap reachable from the entry points associated with the lock ℓ becomes a new disjunct in the resource invariant and the rest of the heap becomes the local state of the thread. Intuitively, when a thread modifies pointers to a heap cell so that it becomes reachable from the entry points associated with a lock, the cell becomes protected by the lock and a part of its resource invariant. In this way, we discover a new disjunct

$$I^1 = \mathtt{h} \Vdash \exists X.\, \mathtt{h.Back} \mapsto X * \mathtt{h.Fwd} \mapsto X * \mathtt{h.Data} \mapsto \_ * X\mathtt{.Back} \mapsto \mathtt{h} * X\mathtt{.Fwd} \mapsto \mathtt{h} * X\mathtt{.Data} \mapsto \_$$

in the resource invariant and a new state p reachable right after line 21. Note that since the variable `n` is a local variable of `thread1`, we existentially quantify its value in I^1. We continue to run the fixed-point computation defined by the underlying sequential heap analysis starting from the state p. The processing of lines 14 and 21 is the same as before, i.e., we use the same approximation I^0 of the resource invariant and get the same disjunct I^1. We stop when the underlying heap analysis reaches a fixed point. One new disjunct I^1 of the resource invariant has been discovered.

We now analyse the code of `thread2`. Whenever the analysis reaches line 29 with an abstract state q, we conjoin the current approximation I^0 of the resource invariant to the state q yielding $q * I^0$ and analyse the code in lines 30–36 starting from this state. This gives us the state

$$q_1 = q * (\mathtt{h} \Vdash \mathtt{h.Back} \mapsto \mathtt{h} * \mathtt{h.Fwd} \mapsto \mathtt{h} * \mathtt{h.Data} \mapsto \_)$$

at line 37. We again take the part of the heap reachable from `h` as a new disjunct in the resource invariant and let the rest of the heap be a new local state of the thread. In this case the new disjunct in the resource invariant is the same as the starting one I^0, so, no new disjuncts in the resource invariant are discovered.

Second iteration. On the previous iteration we found a new disjunct I^1 in the resource invariant associated with the lock ℓ. This means that whenever a thread acquires the lock ℓ, it can get the ownership of a piece of heap with this new shape. To account for this in the analysis we now consider this possibility for all `acquire(`ℓ`)` commands in the program and perform the analysis on the threads starting from the resulting new states. In `thread1` we obtain a new state $p * I^1$ at line 15. The analysis of the code in lines 15–20 in this case gives us the state

$$p_2 = \exists X.\, p * (\mathtt{h}, \mathtt{n} \Vdash \mathtt{h.Back} \mapsto X * \mathtt{h.Fwd} \mapsto \mathtt{n} * \mathtt{h.Data} \mapsto \_ * \\ \mathtt{n.Back} \mapsto \mathtt{h} * \mathtt{n.Fwd} \mapsto X * \mathtt{n.Data} \mapsto \_ * X.\mathtt{Back} \mapsto \mathtt{n} * X.\mathtt{Fwd} \mapsto \mathtt{h} * X.\mathtt{Data} \mapsto \_)$$

at line 21. Again, the part of the heap reachable from `h` forms a new disjunct in the resource invariant. To ensure convergence we abstract it before adding to the resource invariant: the abstraction procedure of the underlying sequential heap analysis abstracts the heap that has two cells `n` and X connected in a doubly-linked list to an arbitrary doubly-linked list giving us a new disjunct in the resource invariant:

$$I^2 = \mathtt{h} \Vdash \exists X, Y.\, \mathtt{h.Back} \mapsto X * \mathtt{h.Fwd} \mapsto Y * \mathtt{h.Data} \mapsto \_ * \mathsf{dll}(Y, \mathtt{h}, \mathtt{h}, X).$$

A similar procedure for `thread2` again gives us the state q_1 at line 37. No new disjuncts in resource invariants are discovered while analysing this thread.

Third iteration. We propagate the newly discovered disjunct I^2 of the resource invariant to `acquire` commands. The new state $p * I^2$ at line 15 gives rise to the state

$$p_3 = \exists X, Y.\, p * (\mathtt{h}, \mathtt{n} \Vdash \mathtt{h.Back} \mapsto X * \mathtt{h.Fwd} \mapsto \mathtt{n} * \mathtt{h.Data} \mapsto \_ * \\ \mathtt{n.Back} \mapsto \mathtt{h} * \mathtt{n.Fwd} \mapsto Y * \mathtt{n.Data} \mapsto \_ * \mathsf{dll}(Y, \mathtt{n}, \mathtt{h}, X))$$

at line 21. Splitting it into the part reachable from `h` and the part unreachable from `h` and abstracting the former again gives us the resource invariant I^2 and the state p. Propagating the new disjunct in the resource invariant to line 29 yields the state

$$q_2 = \exists X, Y.\, q * (\mathtt{h} \Vdash \mathtt{h.Back} \mapsto X * \mathtt{h.Fwd} \mapsto Y * \mathtt{h.Data} \mapsto \_ * \mathsf{dll}(Y, \mathtt{h}, \mathtt{h}, X))$$

at line 37. Splitting this state again does not result in new disjuncts in the resource invariant being discovered.

No new disjuncts in resource invariants were discovered on this iteration, hence, we have reached a fixed point. The resource invariant for the lock ℓ computed by the analysis is given by the set of symbolic heaps $\{I^0, I^1, I^2\}$, i.e., the invariant is $I^0 \vee I^1 \vee I^2$. Furthermore, the program is memory safe.

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{\sigma\}) \sqsubset \top \quad \sigma' \in f_C(\{\sigma\})}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \sigma'}$$

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{\sigma\}) = \top}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \top}$$

$$\frac{(v, \texttt{acquire}(\ell_j), v') \in T \quad j \in \mathsf{Free}(\mathsf{pc}[k : v])}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \sigma}$$

$$\frac{(v, \texttt{release}(\ell_j), v') \in T}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \sigma}$$

Figure 3.2: Operational semantics of concurrent programs with static locks and threads

3.2 Programming language and semantics

Throughout this chapter we fix a program $S = C_1 \parallel \ldots \parallel C_n$ in the concurrent programming language of Section 2.1.4 consisting of n threads $C_1, \ldots, C_n$ that use m locks $\ell_1, \ldots, \ell_m$ for synchronisation. It is technically convenient for us to represent each thread C_k by its CFG $(N_k, T_k, \mathsf{start}_k, \mathsf{end}_k)$ over the set of primitive commands

$$\mathsf{Seq} \cup \{\texttt{acquire}(\ell_k) \mid k = 1..m\} \cup \{\texttt{release}(\ell_k) \mid k = 1..m\},$$

where Seq is a fixed set of primitive sequential commands. Let $N = \bigcup_{k=1}^{n} N_k$ and $T = \bigcup_{k=1}^{n} T_k$ be the set of program points and the control flow relation of the program S, respectively.

As in Section 2.1.4, we assume a concrete separation domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, *, e)$, constructed out of a separation algebra Σ, and monotone and local forward predicate transformers $f_C : D \to D$ for $C \in \mathsf{Seq}$, constructed out of the corresponding local functions on Σ.

The interleaving operational semantics of the program S is defined by a transition relation $\rightarrow_S$ that transforms pairs of program counters (represented by mappings from thread identifiers to program points) and program states:

$$\rightarrow_S : ((\{1, \ldots, n\} \to N) \times \Sigma) \times ((\{1, \ldots, n\} \to N) \times (\Sigma \cup \{\top\})).$$

Note that since the critical regions formed by `acquire` and `release` commands are syntactically scoped in our programming language, we can determine the set $\mathsf{Free}(\mathsf{pc})$ of indices of free locks at every program counter pc, i.e., the set of locks that are not held by any thread. The relation $\rightarrow_S$ is defined by the rules in Figure 3.2. Note that, according to our semantics, primitive sequential commands are executed atomically. Also, a thread that tries to acquire the same lock twice deadlocks.

Let us denote with pc_0 the initial program counter $[1 : \mathsf{start}_1] \ldots [n : \mathsf{start}_n]$ and with pc_f the final one $[1 : \mathsf{end}_1] \ldots [n : \mathsf{end}_n]$. We say that the program S is *safe* when run from an initial state $\sigma_0 \in \Sigma$, if it is not the case that $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \top$ for some program counter pc.

3.3 Constructing thread-modular heap analyses

We now show how to construct a thread-modular heap analysis for the programming language of Section 2.1.4 from a given heap analysis for the sequential subset of the language satisfying certain conditions. As can be seen from the illustrative example in Section 3.1, in our thread-modular heap analysis we have to split abstract heaps into disjoint parts. For this to be possible, the underlying sequential abstract domain has to have a separated structure that allows for performing such splittings. The formal condition sufficient to ensure this is that the abstract domain be a separation domain (Definition 2.2). We therefore specialise the framework of abstract interpretation presented in Section 2.2 to this case. In the setting of Section 3.2, we assume given:

- an abstract separation domain $(D^\sharp, \sqsubseteq, \bigsqcup, \bot, \top, *^\sharp, e^\sharp)$ representing abstract states of the program;
- a monotone concretisation function $\gamma : D^\sharp \rightarrow D$;
- abstract transfer functions $f_C^\sharp : D^\sharp \rightarrow D^\sharp$ defining the abstract semantics of primitive sequential commands $C \in \mathsf{Seq}$

such that

- abstract transfer functions over-approximate the concrete ones:
$$\forall C \in \mathsf{Seq}.\, \forall p \in D^\sharp.\, f_C(\gamma(p)) \sqsubseteq \gamma(f_C^\sharp(p)); \tag{3.1}$$
- the abstract operation of separate combination over-approximates the concrete one:
$$\forall p, q \in D^\sharp.\, \gamma(p) * \gamma(q) \sqsubseteq \gamma(p *^\sharp q). \tag{3.2}$$

Note that we do not require that the abstract transfer functions be local or monotone. We call an abstract interpretation constructed in such a way an *abstract interpretation with state separation.*

We now define a thread-modular analysis on the program $S = C_1 \parallel \ldots \parallel C_n$. To simplify presentation, when defining the analysis in this chapter we assume that the heap at the beginning of the program's execution consists of only initialised shared data structures and, furthermore, that we are given a sequential initialisation routine for the data structure protected by every lock.[1] By running the underlying sequential heap

[1] We lift this simplifying assumption in Chapter 6 when we add dynamic thread creation to our programming language.

$\mathcal{F}^\sharp(I^0)(G^\sharp, I^\sharp) = (\widetilde{G}^\sharp, \widetilde{I}^\sharp)$, where

- $\widetilde{G}^\sharp(\mathsf{start}_k) = e^\sharp$, $k = 1..n$;
- $\widetilde{G}^\sharp(v') = \bigsqcup_{(v,C,v')\in T} g^\sharp_C(G^\sharp(v))$
  for every program point $v' \in N \backslash \{\mathsf{start}_k \mid k = 1..n\}$, where
  $$g^\sharp_C(p) = \begin{cases} f^\sharp_C(p), & \text{if } C \in \mathsf{Seq}; \\ p *^\sharp I^\sharp_k, & \text{if } C \text{ is } \texttt{acquire}(\ell_k); \\ \mathsf{ThreadLocal}_k(p), & \text{if } C \text{ is } \texttt{release}(\ell_k); \end{cases}$$
- $\widetilde{I}^\sharp_k = I^0_k \sqcup \bigsqcup_{(v,\texttt{release}(\ell_k),v')\in T} \mathsf{Protected}_k(G^\sharp(v))$ for every lock ℓ_k.

Figure 3.3: Thread-modular analysis

analysis on every initialisation routine, we can determine an element of the abstract domain $I^0_k \in D^\sharp$ describing the initial state of the data structure protected by every lock ℓ_k, $k = 1..m$. We denote with I^0 the vector of I^0_k, $k = 1..m$.

The main idea of the analysis is to infer the part of the state protected by each lock—its resource invariant. Resource invariants are computed incrementally during the analysis, therefore, for each lock ℓ_k the analysis maintains the current approximation $I^\sharp_k \in D^\sharp$ of the corresponding resource invariant. We denote with $I^\sharp$ the vector of such approximations. In addition, for every program point $v \in N$ in the CFG of a thread the analysis maintains the part $G^\sharp(v) \in D^\sharp$ of the abstract program state owned by the thread at the program point v—its local state. Formally, the analysis operates on the domain $\widehat{D}^\sharp = (N \to D^\sharp) \times (D^\sharp)^m$.

The thread-modular analysis is defined using the functional $\mathcal{F}^\sharp(I^0) : \widehat{D}^\sharp \to \widehat{D}^\sharp$, parameterised by the vector $I^0 \in (D^\sharp)^m$, that takes a pair $(G^\sharp, I^\sharp)$ and produces a pair $(\widetilde{G}^\sharp, \widetilde{I}^\sharp)$ as shown in Figure 3.3. The result of the analysis is defined as a fixed point of the functional $\mathcal{F}^\sharp(I^0)$. According to the definition of $\mathcal{F}^\sharp(I^0)$, the initial local state of every thread is just the abstract denotation of the empty heap $e^\sharp$, in accordance with our simplifying assumption. We take I^0_k as the initial approximation of the resource invariant $I^\sharp_k$. The treatment of sequential commands and control flow in the analysis is standard (see Section 2.2).

The interesting part of the analysis is the treatment of acquiring and releasing locks. When a thread acquires a lock ℓ_k, it obtains the current approximation of the corresponding resource invariant—the current approximation of the resource invariant is $*^\sharp$-conjoined with the current local state of the thread to yield a new local state. This corresponds to the global ACQUIRE axiom of concurrent separation logic. The treatment of `release` mimics the global RELEASE axiom: when a thread releases the lock ℓ_k, its current local state is split into two parts, one of which goes to the resource invariant and the other one becomes the new local state of the thread. The analysis is parameterised by functions $\mathsf{ThreadLocal}_k : D^\sharp \to D^\sharp$ and $\mathsf{Protected}_k : D^\sharp \to D^\sharp$ for each $k = 1..m$ that determine this splitting. The function $\mathsf{ThreadLocal}_k$ determines the part of the state that becomes the local state of the thread and the function $\mathsf{Protected}_k$ the part that goes to the resource invariant. We require that these functions soundly split the state, i.e., that the combination of the parts of the splitting over-approximate the state being split:

$$\forall p \in D^\sharp.\, \gamma(p) \sqsubseteq \gamma(\mathsf{ThreadLocal}_k(p)) * \gamma(\mathsf{Protected}_k(p)). \tag{3.3}$$

A computation of a fixed point of the functional $\mathcal{F}^\sharp(I^0)$ would analyse each thread accumulating possible values of resource invariants during the analysis. Each time a new possible value of a resource invariant associated with a lock is discovered, it would have to be propagated to every `acquire` command for the lock. Hence, each thread is analysed repeatedly, but separately, without exploring the set of interleavings. In this sense the analysis defined by $\mathcal{F}^\sharp(I^0)$ is thread-modular. Note also that after the analysis splits the state at a `release` command, it loses correlations between the parts of the state that become local states of the thread and the parts that go to the resource invariant. This loss of precision is similar to the one observed in thread-modular model checking [30].

3.3.1 Examples of abstract separation domains

Some of the abstract domains presented in Section 2.2.1 can be formulated as abstract separation domains.

Interval domain can be turned into a separation domain by defining $*$ in the following way:

$$\forall p_1, p_2 \in \mathsf{IntDom} \backslash \{\top\}.\, p_1 * p_2 = p_1 \uplus p_2;$$
$$\forall p \in \mathsf{IntDom}.\, p * \top = \top * p = \top.$$

The concrete transfer functions on the domain are local when defined as noted in Section 2.2.1.

Domains based on separation logic (SLL, CDS, DLL) are separation domains with $*$ on symbolic heaps defined as follows:

$$(O_1 \Vdash \exists \vec{X_1}.\, \mathsf{P}_1 \wedge \mathsf{S}_1) * (O_2 \Vdash \exists \vec{X_2}.\, \mathsf{P}_2 \wedge \mathsf{S}_2) = (O_1 * O_2 \Vdash \exists \vec{X_1}, \vec{X_2}.\, (\mathsf{P}_1 \wedge \mathsf{P}_2) \wedge (\mathsf{S}_1 * \mathsf{S}_2))$$

and lifted to the domains pointwise. As noted in Section 2.1.2, the corresponding concrete transfer functions are local.

Domains based on shape graphs can be formulated as separation domains with $*$ defined using the union of shape graphs. However, we cannot use these domains as they are presented in [49, 48], since to define the functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ we have to be able to split shape graphs creating dangling pointers across splittings. The abstract representation of [49, 48] does not allow dangling pointers, but can be extended to allow special *unusable pointers*. These are pointers such that no information about them is preserved by the analysis and dereferencing them results in an error.[2] Adding them preserves the finiteness of the domains. Both dangling pointers resulting from the **delete** command and pointers from the part of the heap computed by $\mathsf{Protected}_k$ into the part of the heap computed by $\mathsf{ThreadLocal}_k$ can be modelled in the abstract representation by unusable pointers.

TVLA is an example of an abstract domain for heap analysis that does not have a straightforward representation as a separation domain with local transfer functions. Thus, we cannot in general use TVLA in our framework.

3.3.2 A heuristic for determining heap splittings

Our thread-modular analysis is parameterised by the functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ that determine heap splittings upon releasing the lock ℓ_k. Any splitting satisfying (3.3) is sound, but choosing a wrong one may lead the analysis to be imprecise. For example, if we transfer a part of the heap that the programmer intended to be owned by a thread to a resource invariant, a later access to this part by the thread will lead the analysis to signal a potential memory-safety violation, represented by the corresponding transfer function returning $\top$. Conversely, if we leave a part of the heap intended to be protected by a lock in the local state of a thread, the analysis will not be able to justify the safety of a later access to this part by another thread holding the lock.

We observe that in many programs the part of the heap protected by a lock can be defined as the set of memory cells reachable via pointers from a given set of variables and that these variables are also protected by the lock. For example, this is the case when a lock protects a cyclic singly-linked list with a sentinel head node pointed to by a given variable. We call such variables the *entry points* for the corresponding lock and denote their set with Entry_k. Assuming that we are given Entry_k for every lock ℓ_k, a reasonable heuristic for determining heap splitting in this situation is defined by $\mathsf{Protected}_k(p)$ computing the part of the heap represented by p reachable from the entry points (recall that in the analysis $\mathsf{Protected}_k$ computes the part of the heap that goes into the resource invariant). The rest of the heap becomes the result of $\mathsf{ThreadLocal}_k$. The intuition is that modifying pointers to a heap cell such that it moves into the part of the heap defined by $\mathsf{Protected}_k$ means that it becomes protected by the corresponding lock

[2] This was suggested to us by Tal Lev-Ami.

and a part of its resource invariant. Conversely when a thread modifies pointers so that a memory cell becomes inaccessible from the entry points of a lock, this signifies that it should be moved from the lock's resource invariant into the thread's local state.

The set of program variables protected by a lock forms a reasonable guess for the set of entry points associated with the lock. The set of entry points can then be inferred using, e.g., tools (both static and dynamic) for analysing correlations between locks and variables that determine the set of locks that are held consistently each time a variable is accessed [63, 73, 17]. In more complex cases the entry points can be given by the user as annotations. Note that the soundness of our analysis does not depend on the particular association of locks and entry points: the analysis can be used with any association.

We now give the definitions of the functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ for the separation domains mentioned in Section 3.3.1.

Interval domain. For $p \in \mathsf{IntDom}\backslash\{\top\}$, $\mathsf{Protected}_k(p)$ just selects the intervals for variables in Entry_k from p, and $\mathsf{ThreadLocal}_k(p)$ returns the rest. Additionally, we let $\mathsf{Protected}_k(\top) = \mathsf{ThreadLocal}_k(\top) = \top$.

Domains based on separation logic. For these domains, instead of computing reachability precisely, $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ compute its approximation—reachability in the formula. We illustrate its definition using SLL as an example.

Let S be the spatial part of a symbolic heap in the SLL domain and U be a set of expressions. Let V be the minimal set of expressions such that

$$U \cup \{F \mid \exists E, \mathsf{S}_1.\, E \in V \wedge \mathsf{S} = H(E,F) * \mathsf{S}_1\} \subseteq V.$$

Here $H(E,F)$ stands for either $E \mapsto F$ or $\mathsf{ls}(E,F)$. We denote the part of S reachable from U with $\mathsf{Reach}(\mathsf{S},U)$ and define it as the $*$-conjunction of the following set of formulae:

$$\{\mathsf{S}_1 \mid \exists E, F, \mathsf{S}_2.\, E \in V \wedge \mathsf{S} = \mathsf{S}_1 * \mathsf{S}_2 \wedge \mathsf{S}_1 = H(E,F)\}.$$

Let $\mathsf{Unreach}(\mathsf{S},U)$ be the formula consisting of all $*$-conjuncts from S that are not in $\mathsf{Reach}(\mathsf{S},U)$.

Consider a symbolic heap $O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}$. To take the equalities in P into account while computing the part of S reachable from the entry points we require that the variables in S be chosen so that for each equivalence class generated by the equalities in P at most one variable from this equivalence class is present in S (with preference given to unquantified variables over quantified ones, and to entry points over other variables). We then define

$$\begin{aligned}\mathsf{Protected}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) = \mathsf{can}(O \cap \mathsf{Entry}_k \Vdash \exists \vec{X}, \vec{Y}.\, (\mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Entry}_k))\\ [\vec{Y}/(\mathsf{vars}(\mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Entry}_k))\backslash\mathsf{Entry}_k)]) \quad (3.4)\end{aligned}$$

and

$$\mathsf{ThreadLocal}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) = \\ \mathsf{can}(O \backslash \mathsf{Entry}_k \Vdash \exists \vec{X}, \vec{Z}.\, (\mathsf{P} \wedge \mathsf{Unreach}(\mathsf{S}, \mathsf{Entry}_k))[\vec{Z}/\mathsf{Entry}_k]) \quad (3.5)$$

for fresh $\vec{Y}, \vec{Z}$ and lift $\mathsf{Protected}_k$ and $\mathsf{ThreadLocal}_k$ to SLL pointwise. Note that, according to this definition, the ownership of the entry points themselves is also transferred into the shared state. It is easy to see that the functions defined in this way satisfy (3.3).

For example, consider a symbolic heap $P = (\mathtt{x}, \mathsf{y}, \mathsf{z} \Vdash \mathsf{ls}(\mathtt{x}, \mathsf{y}) * \mathsf{ls}(\mathsf{y}, \mathtt{NULL}) * \mathsf{z} \mapsto \mathsf{y})$, and assume that $\mathsf{Entry}_k = \{\mathtt{x}\}$. In this case,

$$\mathsf{Protected}_k(P) = \mathsf{can}(\mathtt{x} \Vdash \exists Y.\, \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL})) = (\mathtt{x} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}))$$

and $\mathsf{ThreadLocal}_k(P) = (\mathsf{y}, \mathsf{z} \Vdash \mathsf{z} \mapsto \mathsf{y})$. Note that this splitting breaks the pointer y from the local part of the heap into the shared part: y is a dangling pointer in $\mathsf{ThreadLocal}_k(P)$.

Domains based on shape graphs. In this case, the functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ can be easily defined using reachability in shape graphs.

Note that due to efficiency considerations, for domains more complex than SLL (such as CDS and DLL) the heuristic based on reachability from the entry points is used only to inform design choices in the implementation of $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$. Their implementation for such domains uses incomplete theorem provers for entailments between symbolic heaps, which may result in the splitting computed by $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ differing from the expected one. This may lead the analysis to be overly imprecise (e.g., to signal false memory errors), but preserves soundness, since the analysis can use any functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ satisfying (3.3).

Such flexibility is allowed by the fact that the functions $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ are defined in the abstract domain, not the concrete one. Thus, it is possible for their implementation to compute semantically different splitting for two distinct abstract states representing the same set of concrete states. We can disallow this by requiring that $\mathsf{ThreadLocal}_k : D^\sharp \to D^\sharp$ and $\mathsf{Protected}_k : D^\sharp \to D^\sharp$ over-approximate corresponding functions $\mathsf{ThreadLocal}^\mathsf{c}_k : D \to D$ and $\mathsf{Protected}^\mathsf{c}_k : D \to D$ defined in the concrete domain:

$$\forall p \in D^\sharp.\, \mathsf{ThreadLocal}^\mathsf{c}_k(\gamma(p)) \sqsubseteq \gamma(\mathsf{ThreadLocal}_k(p)) \quad (3.6)$$

and

$$\forall p \in D^\sharp.\, \mathsf{Protected}^\mathsf{c}_k(\gamma(p)) \sqsubseteq \gamma(\mathsf{Protected}_k(p)). \quad (3.7)$$

However, such conditions are not required for the soundness of the analysis. We return to this point in Section 5.6.

We have implemented an instantiation of our framework with the domain DLL in a prototype tool. Preliminary experiments on concurrent heap-manipulating code from Windows device drivers (reported in [34]) confirm the efficiency of the analysis.

The reader may now revisit the example in Section 3.1 to check how it corresponds to the formal definition of the analysis given in this section.

3.4 Soundness

As we noted before, processing of `acquire` and `release` commands in the analysis mimics the corresponding axioms of concurrent separation logic. It is thus tempting to try to justify the soundness of the analysis by showing that its results can be compiled into concurrent separation logic, following Lee et al. [46]. Such a method of proving soundness has an additional benefit that it paves the way for the results of the analysis to be used in theorem proving or proof-carrying code systems [54]. Namely, the proof in the program logic produced by the analysis can be used as a certificate justifying to the code consumer that the result of the analysis is correct. Unfortunately, the thread-modular analysis does not produce proofs in concurrent separation logic, since the resource invariants computed by the analysis are not guaranteed to be precise for the following reasons.

Consider an instantiation of the analysis with one of the abstract domains based on separation logic (Section 3.3.1). As we noted in Section 2.1.1, precision is not preserved by disjunction: $(\mathsf{emp_s} \wedge 10 \mapsto \_) \vee \mathsf{emp}$ is not precise, although $(\mathsf{emp_s} \wedge 10 \mapsto \_)$ and emp are. We use disjunction in the analysis to compute resource invariants, since the join operator in abstract domains based on separation logic over-approximates disjunction. Moreover, the definitions of commonly used inductive predicates include disjunctions that can lead to imprecision: e.g., $\mathtt{h} \Vdash \exists X.\, \mathsf{dll}(\mathtt{h}, X, \mathtt{h}, X)$ is imprecise. The presence of existential quantification in the abstract domain also gives rise to symbolic heaps such as $\mathtt{h} \Vdash \mathsf{ls}(\mathtt{h}, \_)$, which are imprecise. However, note that the resource invariant computed in the example of Section 3.1 is precise.

Thus, although the thread-modular analysis is inspired by concurrent separation logic, this is not its underlying logic. As we show in Section 3.5 below, the analysis can be viewed as generating proofs in a variant of concurrent separation logic without the conjunction rule and the precision restriction. To establish the soundness of the analysis and the corresponding logic without performing two separate proofs, we define a thread-local interpretation of every thread in the program that annotates its program points with elements from the concrete domain describing its local state. This implicitly records the decisions about heap splittings at `release` commands that could be taken by a run of the analysis or a proof in the logic. We show that the thread-local interpretation is sound in a certain sense with respect to the interleaving operational semantics (Section 3.4.1). To prove the soundness of the thread-modular analysis, we then show that its results generate an instance of the thread-local interpretation (Section 3.4.2). To prove the soundness of the logic, we define the notion of validity of Hoare triples for commands with respect to the thread-local interpretation and then prove the soundness of all the proof rules

(Section 3.4.1). In addition, we show that the success of the thread-modular analysis on a program or the provability of a program in our logic implies that the program is data-race free (Section 3.4.4).

3.4.1 Thread-local interpretation and Parallel Decomposition Lemma

In the setting of Section 3.2 we define a *semantic proof* as a triple $(C, G, \mathcal{I})$, where

- C is a command with a CFG $(N, T, \mathsf{start}, \mathsf{end})$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{acquire}(\ell_k) \mid k = 1..m\} \cup \{\mathtt{release}(\ell_k) \mid k = 1..m\}$;
- $G : N \to D$ maps program points of C to semantic annotations;
- $\mathcal{I} \in D^m$ is a vector of resource invariant denotations $\mathcal{I}_k \in D$, $k = 1..m$

such that for all edges $(v, C', v') \in T$

- if $C' \in \mathsf{Seq}$, then
$$f_{C'}(G(v)) \sqsubseteq G(v'); \tag{3.8}$$
- if C' is $\mathtt{acquire}(\ell_k)$, then
$$G(v) * \mathcal{I}_k \sqsubseteq G(v'); \tag{3.9}$$
- if C' is $\mathtt{release}(\ell_k)$, then
$$G(v) \sqsubseteq G(v') * \mathcal{I}_k. \tag{3.10}$$

Note that the elements of D assigned to program points by the semantic annotation mapping G in this definition are similar to label invariants in proof systems for unstructured control flow [22]. Inequalities (3.8), (3.9), and (3.10) are semantic counterparts of the axioms Prim and the global versions of Acquire and Release, respectively. The thread-local interpretation of a command is given by its semantic proof. In Sections 3.4.2 and 3.4.3 we show how to extract semantic proofs for threads in the program from proofs in the logic or results of the analysis.

The core of our proofs of soundness consists of establishing the so-called Separation Property [56]: at any time, the state of the program can be partitioned into that owned by each thread and each free lock. Its formalisation in the original proof of soundness of concurrent separation logic is called the Parallel Decomposition Lemma [12]. The following lemma formalises the property in the case when the local states of threads are defined by semantic proofs, which shows the soundness of our thread-local interpretation with respect to the operational semantics of Section 3.2.

Lemma 3.1 (Parallel Decomposition Lemma). *Assume semantic proofs* $(C_k, G_k, \mathcal{I})$, $k = 1..n$. *If* $\sigma_0 \in \Sigma$ *is such that*

$$\{\sigma_0\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} G_k(\mathsf{start}_k)\right) * \left(\mathop{\circledast}_{k\in\{1,\ldots,m\}} \mathcal{I}_k\right), \tag{3.11}$$

then, whenever $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, *we have*

$$\{\!|\sigma|\!\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} G_k(\mathsf{pc}(k))\right) * \left(\mathop{\circledast}_{k\in\mathsf{Free}(\mathsf{pc})} \mathcal{I}_k\right). \tag{3.12}$$

Proof. We prove the statement of the theorem by induction on the length of the derivation of σ in the operational semantics of the program S. In the base case (3.12) is equivalent to (3.11). Suppose now that

$$\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}[j : v], \sigma \to_S \mathsf{pc}[j : v'], \sigma'.$$

Then $(v, C, v') \in T$ for some atomic command C. We have to show that if

$$\{\sigma\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} G_k((\mathsf{pc}[j : v])(k))\right) * \left(\mathop{\circledast}_{k\in\mathsf{Free}(\mathsf{pc}[j:v])} \mathcal{I}_k\right), \tag{3.13}$$

then

$$\{\!|\sigma'|\!\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} G_k((\mathsf{pc}[j : v'])(k))\right) * \left(\mathop{\circledast}_{k\in\mathsf{Free}(\mathsf{pc}[j:v'])} \mathcal{I}_k\right). \tag{3.14}$$

There are three cases corresponding to the type of the command C.

Case 1. $C \in \mathsf{Seq}$. In this case $\mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v'])$. Let

$$r = \left(\mathop{\circledast}_{\substack{1\le k\le n,\\ k\ne j}} G_k(\mathsf{pc}(k))\right) * \left(\mathop{\circledast}_{k\in W} \mathcal{I}_k\right), \tag{3.15}$$

where $W = \mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v'])$. Then

$$\begin{array}{lll} \{\!|\sigma'|\!\} & \sqsubseteq f_C(\{\sigma\}) & \text{definition of } \to_S \\ & \sqsubseteq f_C(G_j(v) * r) & (3.13) \\ & \sqsubseteq f_C(G_j(v)) * r & f_C \text{ is local} \\ & \sqsubseteq G_j(v') * r & (3.8) \end{array}$$

Case 2. C is $\mathtt{acquire}(\ell_i)$. In this case $i \in \mathsf{Free}(\mathsf{pc}[j : v])$ and $i \notin \mathsf{Free}(\mathsf{pc}[j : v'])$. Let r be defined by (3.15) with $W = \mathsf{Free}(\mathsf{pc}[j : v]) \setminus \{i\} = \mathsf{Free}(\mathsf{pc}[j : v'])$. Then

$$\begin{array}{lll} \{\!|\sigma'|\!\} & \sqsubseteq \{\sigma\} & \text{definition of } \to_S \\ & \sqsubseteq G_j(v) * \mathcal{I}_i * r & (3.13) \\ & \sqsubseteq G_j(v') * r & (3.9) \end{array}$$

Case 3. C is $\mathtt{release}(\ell_i)$. In this case $i \notin \mathsf{Free}(\mathsf{pc}[j : v])$ and $i \in \mathsf{Free}(\mathsf{pc}[j : v'])$. Let r be defined by (3.15) with $W = \mathsf{Free}(\mathsf{pc}[j : v]) = \mathsf{Free}(\mathsf{pc}[j : v']) \setminus \{i\}$. Then

$$\begin{array}{lll} \{\!|\sigma'|\!\} & \sqsubseteq \{\sigma\} & \text{definition of } \to_S \\ & \sqsubseteq G_j(v) * r & (3.13) \\ & \sqsubseteq G_j(v') * \mathcal{I}_i * r & (3.10) \end{array}$$

In all cases we get inequalities equivalent to (3.14), which completes the induction. □

3.4.2 Soundness of the analysis

We are now in a position to state and prove the soundness of the thread-modular analysis of Section 3.3 with respect to the operational semantics of Section 3.2. The following theorem formalises the Separation Property for local states and resource invariants computed by the analysis.

Theorem 3.2 (Soundness of the analysis). *Assume $I^0 \in (D^\sharp)^m$. Let $(G^\sharp, I^\sharp)$ be a fixed point of the functional $\mathcal{F}^\sharp(I^0)$. If $\sigma_0 \in \Sigma$ is such that*

$$\{\sigma_0\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} \gamma(e^\sharp) \right) * \left(\mathop{\circledast}_{k \in \{1,\dots,m\}} \gamma(I^0_k) \right), \tag{3.16}$$

then, whenever $\mathsf{pc}_0, \sigma_0 \rightarrow^*_S \mathsf{pc}, \sigma$, we have

$$\{\!|\sigma|\!\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp(\mathsf{pc}(k))) \right) * \left(\mathop{\circledast}_{k \in \mathsf{Free}(\mathsf{pc})} \gamma(I^\sharp_k) \right). \tag{3.17}$$

Proof. We show that the results of the analysis generate an instance of the thread-local interpretation of Section 3.4.1. For $k = 1..n$ we define $G^\sharp_k$ as the restriction of $G^\sharp$ to program points in N_k and define $\gamma(G^\sharp_k) : N_k \to D$ as follows: $\forall v.\, \gamma(G^\sharp_k)(v) = \gamma(G^\sharp_k(v))$. Let $\mathcal{I}_j = \gamma(I^\sharp_j)$ and let $\mathcal{I}$ be the vector of $\mathcal{I}_j$, $j = 1..m$. We now show that $(C_k, \gamma(G^\sharp_k), \mathcal{I})$ is a semantic proof. Take an edge $(v, C, v') \in T_k$. We consider three cases corresponding to the type of the command C.

For $C \in \mathsf{Seq}$, from (3.1) and the definition of $\mathcal{F}^\sharp(I^0)$ we get

$$f_C(\gamma(G^\sharp_k(v))) \sqsubseteq \gamma(f^\sharp_C(G^\sharp_k(v))) \sqsubseteq \gamma(G^\sharp_k(v')),$$

which is equivalent to (3.8). In the case when C is $\texttt{acquire}(\ell_j)$, from (3.2) we similarly get

$$\gamma(G^\sharp_k(v)) * \gamma(I^\sharp_j) \sqsubseteq \gamma(G^\sharp_k(v) *^\sharp I^\sharp_j) \sqsubseteq \gamma(G^\sharp_k(v')),$$

which is equivalent to (3.9). Finally, for the case when C is $\texttt{release}(\ell_j)$, from (3.3) we get

$$\gamma(G^\sharp_k(v)) \sqsubseteq \gamma(\mathsf{ThreadLocal}_j(G^\sharp_k(v))) * \gamma(\mathsf{Protected}_j(G^\sharp_k(v))) \sqsubseteq \gamma(G^\sharp_k(v')) * \mathcal{I}_j$$

which is equivalent to (3.10).

Thus, $(C_k, \gamma(G^\sharp_k), \mathcal{I})$, $k = 1..n$ are indeed semantic proofs. By the definition of $\mathcal{F}^\sharp(I^0)$ we have $G^\sharp_k(\mathsf{start}_k) = e^\sharp$ and $I^0_j \sqsubseteq I^\sharp_j$. Hence, from (3.16) it follows that

$$\{\sigma_0\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} \gamma(e^\sharp) \right) * \left(\mathop{\circledast}_{k \in \{1,\dots,m\}} \gamma(I^0_k) \right) \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp_k(\mathsf{start}_k)) \right) * \left(\mathop{\circledast}_{k \in \{1,\dots,m\}} \gamma(I^\sharp_k) \right).$$

Applying Lemma 3.1, we then get

$$\{\!|\sigma|\!\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp_k(\mathsf{pc}(k))) \right) * \left(\mathop{\circledast}_{k \in \mathsf{Free}(\mathsf{pc})} \gamma(I^\sharp_k) \right)$$

as required. □

The proof of the theorem allows us to justify the requirements we imposed in the construction of the thread-modular analysis in Section 3.3: (3.1), (3.2), and (3.3) ensure correct processing of sequential commands, `acquire`, and `release`, respectively.

Note that not only does Theorem 3.2 restrict initial and final states of the program, but it also provides information about states at the intermediate points in the computation (and so do soundness theorems for other analyses in this dissertation). This is because, unlike triples in Hoare logics, fixed points computed by analyses carry information about intermediate states, which can be used, e.g., to enable compiler optimisations.

We can reformulate the soundness statement given by Theorem 3.2 in a way that is more usual for program analyses based on abstract interpretation (Section 2.2). Let $\widehat{D} = N^n \to D$ and let $R \in \widehat{D}$ be the collecting semantics of the program S:

$$R(\mathsf{pc}) = \bigsqcup \{\sigma \mid \mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma \text{ and } \sigma_0 \text{ satisfies (3.16)}\}.$$

Following the statement of the theorem, we can define a concretisation function $\hat{\gamma} : \widehat{D}^\sharp \to \widehat{D}$ for the results of the thread-modular analysis as follows:

$$\hat{\gamma}(G^\sharp, I^\sharp)(\mathsf{pc}) = \left(\mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp(\mathsf{pc}(k))) \right) * \left(\mathop{\circledast}_{k \in \mathsf{Free}(\mathsf{pc})} \gamma(I_k^\sharp) \right).$$

Then inequality (3.17) can be rewritten as

$$R \sqsubseteq \hat{\gamma}(G^\sharp, I^\sharp),$$

which shows that the collecting semantics is over-approximated by the concretisation of the results of the analysis.

The soundness statement can be used to check safety properties as described in Section 2.2. In particular, we can check memory safety as follows. As was noted in Section 2.1, we usually use the greatest element $\top$ of the concrete domain to denote an error state after a memory-safety violation. If the concretisation function γ satisfies

$$\forall p \in D^\sharp.\, \gamma(p) = \top \Rightarrow p = \top, \tag{3.18}$$

then $\top$ in the abstract domain is the unique pre-image under γ of $\top$ in the concrete domain and, hence, can also be used to denote the error state. We therefore say that the analysis succeeds if its result $(G^\sharp, I^\sharp)$ is such that $G^\sharp(v) \sqsubset \top$ for every program point v and $I_k^\sharp \sqsubset \top$ for every lock ℓ_k. We denote this with $(G^\sharp, I^\sharp) \sqsubset \top$. As an easy consequence of Theorem 3.2, we get that the success of the analysis implies that the program analysed is memory safe.

Corollary 3.3 (Memory safety). *Under the conditions of Theorem 3.2, if the concretisation function γ satisfies (3.18) and $(G^\sharp, I^\sharp) \sqsubset \top$, then the program S is safe when run from initial states σ_0 satisfying (3.16).*

3.4.3 Soundness of the logic

We assume the setting of Sections 2.1.3 and 2.1.4.

Theorem 3.4 (Soundness of the logic). *Assume $I \vdash \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$, where either*

- *the resource invariants in I are precise and the $*$ operation is cancellative; or*
- Conj *is not used in the derivation of the triples.*

Then for any $\sigma_0 \in \Sigma$ such that

$$\sigma_0 \in \left(\mathop{\circledast}_{k=1}^{n} [\![P_k]\!]\right) * \left(\mathop{\circledast}_{k=1}^{m} [\![I_k]\!]\right), \tag{3.19}$$

the program S is safe when run from σ_0, and whenever $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}_\mathsf{f}, \sigma$, we have

$$\sigma \in \left(\mathop{\circledast}_{k=1}^{n} [\![Q_k]\!]\right) * \left(\mathop{\circledast}_{k=1}^{m} [\![I_k]\!]\right). \tag{3.20}$$

To prove Theorem 3.4, we first define a notion of validity of Hoare triples with respect to the thread-local interpretation of Section 3.4.1 and prove the soundness of the proof rules in this interpretation. Soundness of the logic with respect to the concrete semantics is then an easy consequence of Lemma 3.1.

Definition 3.5. *$I \vDash \{P\}\ C\ \{Q\}$ if there exists a semantic proof $(C, G, [\![I]\!])$ such that $G(\mathsf{start}) \sqsubseteq [\![P]\!]$ and $G(\mathsf{end}) \sqsubseteq [\![Q]\!]$, where start and end are the starting and the final program points of the CFG of C.*

We say that an inference rule is sound with respect to the thread-local interpretation if it preserves validity of judgements (as defined by the relation $\vDash$ above).

Lemma 3.6. *The axioms* Prim, Acquire, *and* Release *are sound with respect to the thread-local interpretation.*

Lemma 3.7. *The rules* Seq, Choice, Loop, *and* Conseq *are sound with respect to the thread-local interpretation.*

We omit straightforward proofs of Lemmas 3.6 and 3.7 and proceed to prove the soundness of the rules Frame, Disj, and Conj. To this end, we show that we can construct semantic proofs for the conclusions of these rules out of semantic proofs for their premisses. This is essentially a semantic counterpart of a standard proof that these rules are admissible in the logic including the global Acquire and Release axioms, i.e., that a derivation using these rules can be converted into a derivation that does not use them. By using semantic proofs instead of derivations in our proof system, we avoid having to deal with the syntactic form of the proof rules in the logic and the control-flow constructs in our programming language.

Lemma 3.8. *(i) For any $r \in D$, if $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G(v) * r$.*

(ii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \sqcup G_2(v)$.

(iii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \sqcap G_2(v)$, provided the resource invariant denotations in $\mathcal{I}$ are precise and the $*$ operation is cancellative.

Proof. Consider an edge (v, C', v') in the CFG of the command C. When $C' \in \mathsf{Seq}$, inequality (3.8) follows from the fact that the predicate transformer $f_{C'}$ is local, and distributes over $\sqcup$ and $\sqcap$ (the latter is true by construction of transformers defined by pointwise lifting from Σ). We omit the easy case when C' is $\texttt{acquire}(\ell_k)$. Suppose now that C' is $\texttt{release}(\ell_k)$. We consider every case in the lemma in turn.

(i) Using the definition of G', we get

$$G'(v) = G(v) * r \sqsubseteq G(v') * \mathcal{I}_k * r = G'(v') * \mathcal{I}_k.$$

(ii) The $*$ operation distributes over $\sqcup$:

$$\forall p_1, p_2, q \in D.\, (p_1 \sqcup p_2) * q = (p_1 * q) \sqcup (p_2 * q).$$

Hence,

$$G'(v) = G_1(v) \sqcup G_2(v) \sqsubseteq (G_1(v') * \mathcal{I}_k) \sqcup (G_2(v') * \mathcal{I}_k) = (G_1(v') \sqcup G_2(v')) * \mathcal{I}_k = G'(v') * \mathcal{I}_k.$$

(iii) It is easy to check that if $*$ is cancellative, then for a precise $q \in D$ and any $p_1, p_2 \in D$ we have

$$(p_1 \sqcap p_2) * q = (p_1 * q) \sqcap (p_2 * q). \tag{3.21}$$

Thus, in this case we get

$$G'(v) = G_1(v) \sqcap G_2(v) \sqsubseteq (G_1(v') * \mathcal{I}_k) \sqcap (G_2(v') * \mathcal{I}_k) = (G_1(v') \sqcap G_2(v')) * \mathcal{I}_k = G'(v') * \mathcal{I}_k.$$

In all cases we get (3.10), which completes the proof. □

Corollary 3.9. *The rules* FRAME *and* DISJ *are sound with respect to the thread-local interpretation. So is* CONJ *when the resource invariants in I are precise and the $*$ operation is cancellative.*

Lemma 3.10. *If $I \vdash \{P\}\ C\ \{Q\}$ and the restrictions on the derivation from Theorem 3.4 hold, then $I \vDash \{P\}\ C\ \{Q\}$.*

The proof is by induction on the derivation of $I \vdash \{P\}\ C\ \{Q\}$ using Lemmas 3.6 and 3.7 and Corollary 3.9.

Proof of Theorem 3.4. By Lemma 3.10, $I \models \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$, hence, by Definition 3.5, there exist semantic proofs $(C_k, G_k, [\![I]\!])$, $k = 1..n$ such that $G_k(\mathsf{start}_k) \sqsubseteq [\![P_k]\!]$ and $G_k(\mathsf{end}_k) \sqsubseteq [\![Q_k]\!]$. Consider a state σ_0 satisfying (3.19). Let $\mathcal{I} = [\![I]\!]$ in Lemma 3.1, then (3.11) is fulfilled. We have $G_k(\mathsf{end}_k) \sqsubset \top$, from which it follows that $\forall v.\, G_k(v) \sqsubset \top$. Therefore, for any pc and σ such that $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, from (3.12) we get $\{\!|\sigma|\!\} \sqsubset \top$, i.e., S is safe when run from σ_0. Now letting $\mathsf{pc} = \mathsf{pc_f}$ and using (3.12), we get (3.20). $\Box$

As follows from Theorem 2.4, Lemma 3.10, and hence, Theorem 3.4 also hold for the specialisation of the logic to the domain RAM (Section 2.1.3), i.e., when $D = \mathsf{RAM}$, $\mathsf{Seq} = \mathsf{SeqRAM}$, and proofs use the syntactic versions of the axiom Prim from Figure 2.5.

Reynolds's counterexample. Note that (3.21) does not hold in general for imprecise q, thus, the proof of Lemma 3.8 may not be extended to show the soundness of the conjunction rule in the case of imprecise resource invariants. The intuitive reason for the unsoundness of the conjunction rule in this case is that imprecise resource invariants allow splitting the heap at a `release` command in different ways in different branches of the proof. Thus, in the two premises of Conj there may be different understanding of what the splitting of the global heap into thread-local and protected parts should be. Trying to $\wedge$-conjoin two such judgements about the local state of a thread then leads to inconsistency. For completeness, we now reproduce Reynolds's counterexample demonstrating the unsoundness [56]. Consider the instantiation of the logic for the domain RAM and a program consisting of a single thread executing the code

$$\mathtt{acquire}(\ell);\ \mathtt{skip};\ \mathtt{release}(\ell)$$

We denote with one the assertion $\mathsf{emp_s} \wedge 10 \mapsto \_$. Let $I_\ell = \mathsf{emp_s}$. First, we have the following derivation (we elide the premises of the rule of consequence dealing with implications between assertions):

$$\dfrac{\dfrac{\dfrac{}{I \vdash \{\mathsf{emp_s}\}\ \mathtt{skip}\ \{\mathsf{emp_s}\}}\ \textsc{Skip}}{I \vdash \{(\mathsf{emp} \vee \mathsf{one}) * \mathsf{emp_s}\}\ \mathtt{skip}\ \{\mathsf{emp} * \mathsf{emp_s}\}}\ \textsc{Conseq}}{I \vdash \{\mathsf{emp} \vee \mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp}\}}\ \textsc{Acquire, Release, Seq}$$

Then, from the conclusion of this proof, we can construct two derivations:

$$\dfrac{\dfrac{I \vdash \{\mathsf{emp} \vee \mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp}\}}{I \vdash \{(\mathsf{emp} \vee \mathsf{one}) * \mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp} * \mathsf{one}\}}\ \textsc{Frame}}{I \vdash \{\mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{one}\}}\ \textsc{Conseq}$$

and

$$\dfrac{I \vdash \{\mathsf{emp} \vee \mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp}\}}{I \vdash \{\mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp}\}}\ \textsc{Conseq}$$

Applying the conjunction rule and simplifying using the rule of consequence, we get:

$$\frac{I \vdash \{\mathsf{one} \wedge \mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathsf{emp} \wedge \mathsf{one}\}}{I \vdash \{\mathsf{one}\}\ \mathtt{acquire}(\ell); \mathtt{skip}; \mathtt{release}(\ell)\ \{\mathbf{false}\}}\ \textsc{Conseq}$$

For this proof, (3.19) is satisfied by a state σ_0 with the empty stack and the heap containing an allocated cell at the address 10. However, (3.20) is not satisfied by any state σ, even though $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}_\mathsf{f}, \sigma_0$. Hence, the soundness of the logic is violated. The reason is that here we are combining two derivations of which the first one says that the cell at the address 10 is kept by the thread and the second one that its ownership is transferred to the part of the heap protected by the lock.

Logical variables. Assume that Σ is an algebra with logical variables, i.e., $\Sigma = \Sigma' \times \text{Ints}$, and that the functions f_C for $C \in \mathsf{Seq}$ are lifted from functions on Σ' (Section 2.1.3). In this case, we can add to the logic the rules Exists and Forall. We say that $p \in D$ *does not depend on the interpretation of logical variables*, if $p = \top$, or $p \neq \top$ and for any $(\sigma', \mathbf{i}) \in p$ and $\mathbf{i}' \in \text{Ints}$ we have $(\sigma', \mathbf{i}') \in p$. For the rules to be sound in concurrent setting, we have to require that the denotations of resource invariants do not depend on interpretations (in the case of the domain RAM this is satisfied if resource invariants do not contain free logical variables). For the soundness of Forall, we have to require additionally that the resource invariants be precise and the $*$ operation be cancellative.

For a logical variable X let $\mathsf{Exists}(X) : D \to D$, respectively, $\mathsf{Forall}(X) : D \to D$ be the semantic counterparts of existential, respectively, universal quantification of X, defined as follows:

$$\mathsf{Exists}(X, p) = \begin{cases} \{(\sigma', \mathbf{i}) \mid \exists u \in \text{Values}.\, (\sigma', \mathbf{i}[X : u]) \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top; \end{cases}$$

$$\mathsf{Forall}(X, p) = \begin{cases} \{(\sigma', \mathbf{i}) \mid \forall u \in \text{Values}.\, (\sigma', \mathbf{i}[X : u]) \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top. \end{cases}$$

The proof of soundness of Exists and Forall with respect to the thread-local interpretation is done by establishing the following analogue of Lemma 3.8.

Lemma 3.11. *Under the above conditions, for any logical variable X and resource invariants $\mathcal{I}$ that do not depend on interpretations:*

(i) If $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = \mathsf{Exists}(X, G(v))$.

(ii) If $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = \mathsf{Forall}(X, G(v))$, provided the resource invariant denotations in $\mathcal{I}$ are precise and the $*$ operation is cancellative.

The proof is similar to that of Lemma 3.8.

It follows that Theorem 3.4 holds for the logic extended with the rules Exists and Forall, subject to the conditions given above.

3.4.4 Data-race freedom

Consider the case when Σ = States, D = RAM (Section 2.1.1) and Seq = SeqRAM (Section 2.1.2). We prove that in this case the success of the thread-modular analysis on a program or the provability of a program in our logic implies that the program has no data races (both on stack variables and on heap cells).

For a state $\sigma \in \Sigma$ let $\mathsf{accesses}(C, \sigma)$, respectively, $\mathsf{writes}(C, \sigma)$ be the set of variables and locations that a primitive sequential command $C \in \mathsf{SeqRAM}$ may access (i.e., read, write, or dispose), respectively, write to or dispose, when run from the state σ according to the semantics of commands SeqRAM defined in Figure 2.3.

Definition 3.12 (Interfering commands). *Commands C' and C'' from* SeqRAM *interfere with each other when executed from the state σ, denoted with $C' \bowtie_\sigma C''$, if*

$$\mathsf{accesses}(C', \sigma) \cap \mathsf{writes}(C'', \sigma) \neq \emptyset$$

or

$$\mathsf{writes}(C', \sigma) \cap \mathsf{accesses}(C'', \sigma) \neq \emptyset.$$

Given this formulation of interference, the usual notion of data races is formulated as follows.

Definition 3.13 (Data race). *The program S has a data race when run from an initial state $\sigma_0 \in \Sigma$ if for some i, j, and* pc *such that $i \neq j$, $\mathsf{pc}(i) = v_i$, and $\mathsf{pc}(j) = v_j$ and state $\sigma \in \Sigma$ such that $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, there exist CFG edges $(v_i, C', v_i') \in T_i$ and $(v_j, C'', v_j') \in T_j$ in the control-flow relations of threads i and j labelled with commands C' and C'' from* SeqRAM *such that*

$$C', \sigma \not\rightsquigarrow \top; \quad C'', \sigma \not\rightsquigarrow \top; \quad C' \bowtie_\sigma C''. \tag{3.22}$$

We first prove that the existence of an instance of the thread-local interpretation for a program (as defined in Lemma 3.1) such that all local states and resource invariants are distinct from $\top$ implies that the program is data-race free.

Lemma 3.14. *Under the conditions of Lemma 3.1 with D =* RAM *and* $\mathsf{Seq} = \mathsf{SeqRAM}$, *if $(G, \mathcal{I}) \sqsubset \top$, where $G = \biguplus_{k=1}^n G_k$, then the program S has no data races when run from initial states σ_0 satisfying (3.11).*

Proof. Suppose the contrary: there exist i, j, and pc such that $i \neq j$, $\mathsf{pc}(i) = v_i$, and $\mathsf{pc}(j) = v_j$, a state σ such that $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, CFG edges $(v_i, C', v_i') \in T_i$, and $(v_j, C'', v_j') \in T_j$ labelled with commands C' and C'' from SeqRAM such that (3.22) holds.

By Lemma 3.1, $\sigma \in r * G(v_i) * G(v_j)$ for some r. Hence,

$$\sigma = \sigma_0 * \sigma_1 * \sigma_2, \tag{3.23}$$

where

$$\sigma_0 \in r, \quad \sigma_1 \in G(v_i), \quad \sigma_2 \in G(v_j). \tag{3.24}$$

Since $(G, \mathcal{I}) \sqsubset \top$, it follows that $f_{C'}(G(v_i)) \sqsubset \top$ and $f_{C''}(G(v_j)) \sqsubset \top$. From this and (3.24) we obtain $f_{C'}(\sigma_1) \sqsubseteq f_{C'}(G(v_i)) \sqsubset \top$. So, $f_{C'}(\sigma_1) \sqsubset \top$ and, analogously, $f_{C''}(\sigma_2) \sqsubset \top$. Hence, $C', \sigma_1 \not\rightsquigarrow \top$ and $C'', \sigma_2 \not\rightsquigarrow \top$. From this and the fact that $C' \bowtie_\sigma C''$ using the definition of $*$ and the predicate transformers for primitive sequential commands, we easily get that $\sigma_1 * \sigma_2$ is undefined, which contradicts (3.23). The intuition behind this is that from $C', \sigma_1 \not\rightsquigarrow \top$ and $C'', \sigma_2 \not\rightsquigarrow \top$ it follows that both σ_1 and σ_2 should have the full permission for the same variable or location accessed by C' and C'', which makes the state $\sigma_1 * \sigma_2$ inconsistent. □

As corollaries of Lemma 3.14, we easily get data-race freedom theorems for our logic and analysis.

Corollary 3.15 (Data-race freedom: analysis). *Under the conditions of Theorem 3.2 with $D = \mathsf{RAM}$ and $\mathsf{Seq} = \mathsf{SeqRAM}$, if the concretisation function γ satisfies (3.18) and $(G^\sharp, I^\sharp) \sqsubset \top$, then the program S has no data races when run from initial states σ_0 satisfying (3.16).*

Corollary 3.16 (Data-race freedom: logic). *Under the conditions of Theorem 3.4 with $D = \mathsf{RAM}$ and $\mathsf{Seq} = \mathsf{SeqRAM}$, the program S has no data races when run from initial states σ_0 satisfying (3.19).*

3.5 Certificate generation

We now show that the thread-modular analysis of Section 3.3 can be viewed as generating proofs in a variant of concurrent separation logic without the conjunction rule and the precision restriction. Similar correspondence can be established for other pairs of logics and analyses in this dissertation. We present a detailed construction only in this section. The details of proof generation follow the existing work on proof-producing program analysis for simpler domains [74].

In the setting of Sections 2.1.3 and 2.1.4 We assume an assertion language $\mathcal{L}$ for denoting elements of the concrete separation domain D distinct from $\top$, equipped with a proof system for deriving tautological assertions, i.e., assertions denoting Σ. In the case when $D = \mathsf{RAM}$, we can use the assertion language presented in Section 2.1.1 and an adaptation of the proof systems from [66] or [3]. We assume that the assertion language includes implication $\Rightarrow$, separating conjunction $*$, and the assertion emp with the expected interpretation. We assume that the proof system for the assertion language includes a rule for the transitivity of implication and the following rules:

$$\frac{P_1 \Rightarrow P_2 \quad Q_1 \Rightarrow Q_2}{P_1 * P_2 \Rightarrow Q_1 * Q_2} \tag{3.25}$$

$$\frac{}{\mathsf{emp} * P \Leftrightarrow P} \tag{3.26}$$

We consider a variant of the logic with a set of axioms for primitive commands formulated using the assertion language $\mathcal{L}$ that are sound with respect to the concrete predicate transformers f_C. For the case of the domain RAM, we can take the axioms for primitive commands SeqRAM in Figure 2.5.

We make several assumptions about the ingredients of the analysis. First, we assume that the analysis is equipped with a translation function $\mathsf{tr} : D^\sharp \backslash \{\top\} \to \mathcal{L}$ that for a given abstract state $\mathsf{tr}(p)$ defines a formula in the language $\mathcal{L}$ representing its concretisation: $[\![\mathsf{tr}(p)]\!] = \gamma(p)$. We assume further that the analysis is equipped with a proof-producing component that constructs proofs of the following judgements in the formal system for $\mathcal{L}$:

$$\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(p)\}\ C\ \{\mathsf{tr}(f^\sharp_C(p))\}, \tag{3.27}$$

$$\mathsf{tr}(p) \Rightarrow \mathsf{tr}(p \sqcup q), \tag{3.28}$$

$$\mathsf{tr}(p) * \mathsf{tr}(q) \Rightarrow \mathsf{tr}(p *^\sharp q), \tag{3.29}$$

$$\mathsf{tr}(p) \Rightarrow \mathsf{tr}(\mathsf{Protected}_k(p)) * \mathsf{tr}(\mathsf{ThreadLocal}_k(p)),\ \ k = 1..m \tag{3.30}$$

for any $p, q \in D^\sharp$, $I^\sharp \in (D^\sharp)^m$, and $C \in \mathsf{Seq}$. Note that the formulae and the triple are semantically valid, as follows from (3.1), (3.2), and (3.3). The implementation of transfer functions and the join operator in abstract domains based on separation logic is usually done via proof search in a particular proof system [3, 81]. This search procedure can thus be instrumented to construct the required proofs.

We now show that, under the above assumptions, we can construct a proof of the program in our logic with assertions in $\mathcal{L}$ from a successful run of the thread-modular analysis. Let $(G^\sharp, I^\sharp)$ be the result of the analysis as defined in Section 3.3. We first construct derivations of the triples

$$\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(e^\sharp)\}\ C_k\ \{\mathsf{tr}(G^\sharp(\mathsf{end}_k))\},\ \ k = 1..n. \tag{3.31}$$

We first show that for any CFG edge $(v, C, v') \in T$ we can derive

$$\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(G^\sharp(v))\}\ C\ \{\mathsf{tr}(G^\sharp(v'))\}. \tag{3.32}$$

We consider three cases depending on the command C.

Case 1. C is a primitive command. Then $G^\sharp(v') = f^\sharp_C(G^\sharp(v)) \sqcup p$ for some $p \in D^\sharp$ and the derivation is constructed as follows:

$$\dfrac{\dfrac{}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(G^\sharp(v))\}\ C\ \{\mathsf{tr}(f^\sharp(G^\sharp(v)))\}}\,(3.27) \qquad \dfrac{}{\mathsf{tr}(f^\sharp(G^\sharp(v))) \Rightarrow \mathsf{tr}(G^\sharp(v'))}\,(3.28)}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(G^\sharp(v))\}\ C\ \{\mathsf{tr}(G^\sharp(v'))\}}\,\textsc{Conseq}$$

Case 2. C is `acquire`(ℓ_j)*.* Then $G^\sharp(v') = (I^\sharp_j *^\sharp G^\sharp(v)) \sqcup p$ for some $p \in D^\sharp$ and, hence, we can derive

$$\dfrac{\dfrac{}{\mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{tr}(I^\sharp_j *^\sharp G^\sharp(v))}\,(3.29) \qquad \dfrac{}{\mathsf{tr}(I^\sharp_j *^\sharp G^\sharp(v)) \Rightarrow \mathsf{tr}(G^\sharp(v'))}\,(3.28)}{\mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{tr}(G^\sharp(v'))}$$

where the last rule used is the transitivity of implication. Denoting the above derivation with (*), the derivation of the required triple is as follows:

$$\dfrac{(**) \qquad \dfrac{\dfrac{}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{emp}\}\ \mathtt{acquire}(\ell_j)\ \{\mathsf{tr}(I^\sharp_j)\}}\,\textsc{Acquire}}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{emp} * \mathsf{tr}(G^\sharp(v))\}\ \mathtt{acquire}(\ell_j)\ \{\mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v))\}}\,\textsc{Frame} \qquad (*)}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(G^\sharp(v))\}\ \mathtt{acquire}(\ell_j)\ \{\mathsf{tr}(G^\sharp(v'))\}}\,\textsc{Conseq}$$

where (**) is $\mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{emp} * \mathsf{tr}(G^\sharp(v))$, derived from (3.26).

Case 3. C is `release`(ℓ_j)*.* Then $G^\sharp(v') = \mathsf{ThreadLocal}_j(G^\sharp(v)) \sqcup p$ and $I^\sharp_j = \mathsf{Protected}_j(G^\sharp(v)) \sqcup q$ for some $p, q \in D^\sharp$. Hence, we can derive

$$\dfrac{}{\mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{tr}(\mathsf{Protected}_j(G^\sharp(v))) * \mathsf{tr}(\mathsf{ThreadLocal}_j(G^\sharp(v)))}\,(3.30)$$

and

$$\dfrac{\dfrac{}{\mathsf{tr}(\mathsf{Protected}_j(G^\sharp(v))) \Rightarrow \mathsf{tr}(I^\sharp_j)}\,(3.28) \qquad \dfrac{}{\mathsf{tr}(\mathsf{ThreadLocal}_j(G^\sharp(v))) \Rightarrow \mathsf{tr}(G^\sharp(v'))}\,(3.28)}{\mathsf{tr}(\mathsf{Protected}_j(G^\sharp(v))) * \mathsf{tr}(\mathsf{ThreadLocal}_j(G^\sharp(v))) \Rightarrow \mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v'))}\,(3.25)$$

From the last two derivations by the transitivity of implication we get $\mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v'))$. Denoting the derivation of this fact with (*), the required triple is derived as follows:

$$\dfrac{(*) \qquad \dfrac{\dfrac{}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(I^\sharp_j)\}\ \mathtt{release}(\ell_j)\ \{\mathsf{emp}\}}\,\textsc{Release}}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(I^\sharp_j) * \mathsf{tr}(G^\sharp(v'))\}\ \mathtt{release}(\ell_j)\ \{\mathsf{emp} * \mathsf{tr}(G^\sharp(v'))\}}\,\textsc{Frame} \qquad (**)}{\mathsf{tr}(I^\sharp) \vdash \{\mathsf{tr}(G^\sharp(v))\}\ \mathtt{release}(\ell_j)\ \{\mathsf{tr}(G^\sharp(v'))\}}\,\textsc{Conseq}$$

where (**) is $\mathsf{emp} * \mathsf{tr}(G^\sharp(v)) \Rightarrow \mathsf{tr}(G^\sharp(v))$, derived from (3.26).

Given the derivations of triples (3.31), it is easy to construct derivations for triples (3.32) using the rules for control-flow constructs—Seq, Choice, and Loop. This completes certificate generation.

3.6 Related work

The heap analysis for concurrent programs presented by Yahav [80] relies on abstracting program interleaving and thus does not scale well.

A number of analyses have been developed to detect races in multithreaded programs, both automatic (e.g., [73, 17, 53, 63, 52]) and requiring user annotations (e.g., [29, 9, 36]). To the best of our knowledge all of the automatic tools are either overly imprecise or unsound in the presence of deep heap update. The analyses that require annotations, which are usually based on type systems, preclude ownership transfer; besides, the annotations required by them are often too heavyweight. In contrast, our analysis handles ownership transfer and requires lightweight annotations that can be inferred by existing automatic tools. Some of the techniques for race detection (including [63, 73, 17]) provide information about which locks protect which variables. Such techniques are complementary to ours—they can be used to discover entry points for resource invariants needed by our analysis.

Since the thread-modular analysis presented in this chapter was published [34], there have been several proposals of alternative analyses for heap-manipulating concurrent programs.

Vafeiadis [76] proposed an analysis inspired by ours, but based on RGSep [77]—a combination of rely-guarantee reasoning and separation logic that uses relations, rather than invariants, to describe interference (discussed further in Section 4.7). The logic and the analysis can be used to reason modularly about fine-grained and non-blocking concurrent programs, which typically do not satisfy O'Hearn's Ownership Hypothesis.[3]

Manevich et al. [50, 5] proposed an alternative approach to modular analysis for fine-grained and non-blocking concurrency, based on the notion of heap decomposition. Their analysis maintains several views of the global heap, each giving precise information about the local state of a thread and the shared state, and very coarse information about the rest of the heap. In contrast with our analysis and the one by Vafeiadis [76], different views may maintain information about overlapping parts of the heap. At every step of the analysis, several views of the global leap are combined to yield a more precise view, the command of a thread is executed on the combination, and the result is again decomposed into multiple views. Whereas an analysis based on the approach described here signals a false error if the splitting of the heap is guessed incorrectly, the analysis of Manevich et al. [50] in this situation becomes less efficient, in the worst case degenerating into interleaving enumeration. The analysis is based on TVLA [72] and is parameterised with predicates that TVLA uses to compute views and information defining the composition of views for every command. Appropriate instantiations have been devised for certain fine-

[3] Fine-grained synchronisation protects different parts of the same shared data structure with different locks. Non-blocking synchronisation does not use locks at all, relying instead on alternative low-level synchronisation techniques, usually, atomic compare-and-swap (CAS) instructions.

```
DATA buf;                          get(DATA *y) {
bool full = false;                   bool flag = true;
LOCK ℓ;
                                     while (flag) {
put(DATA x) {                          acquire(ℓ);
  bool flag = true;                    if (full) {
                                         *y = buf;
  while (flag) {                         full = false;
    acquire(ℓ);                          flag = false;
    if (!full) {                       }
      buf = x;                         release(ℓ);
      full = true;                   }
      flag = false;                }
    }
    release(ℓ);
  }
}
```

Figure 3.4: Definitions of put and get

```
DATA x;      ||  DATA y;               DATA x;      ||
x = new();   ||  get(&y);              x = new();   ||  DATA y;
put(x);      ||  delete y;             put(x);      ||  get(&y);
                                       delete x;    ||

           (a)                                    (b)
```

Figure 3.5: A single-element buffer (a) with ownership transfer (b) without ownership transfer. The operations put and get are defined in Figure 3.4.

grained and non-blocking algorithms, however, there are no instantiations available for coarse-grained programs transferring the ownership of memory cells between threads, such as the one in Figure 3.1. In contrast, the analysis presented here is parameterised with heuristics determining ownership transfer of memory cells among threads; the splitting into local and shared heap is then computed automatically. In summary, the approach taken by Manevich et al. [50] yields analyses that are more flexible than the one presented here, but less efficient and requiring more user input.

The fact that our analysis uses a pre-defined heuristic to split the heap at every release command may lead it to signal false memory errors. Consider the two programs in Figure 3.5 (adapted from [56]), where the global variables and the operations put and get are defined in Figure 3.4. We can prove that threads C in both programs satisfy $\{\mathsf{emp}\}\ C\ \{\mathsf{emp}\}$, hence, the programs are memory safe. However, although the programs are similar, their proofs require different resource invariants for the lock ℓ: $\mathtt{buf}, \mathtt{full} \Vdash (\mathtt{full} \wedge \mathtt{buf} \mapsto \_) \vee (\neg \mathtt{full} \wedge \mathsf{emp}_\mathbf{h})$ for program (a) and $\mathtt{buf}, \mathtt{full} \Vdash \mathsf{emp}_\mathbf{h}$ for program (b). Intuitively, in (a) the first thread transfers the ownership of the cell allocated at the address `x` to the second thread via the buffer, and in (b) it does not. In O'Hearn's words, "ownership is in the eye of the asserter" [56]:

> Transfer of ownership is not something that is determined operationally. Whether we transfer the storage associated with a single address, a segment, a linked list, or a tree depends on what we want to prove. Of course, what we can prove is constrained by code outside of critical regions.

In particular, ownership is a global property of the program. Since our analysis decides on ownership transfer locally at every `release` command, for any choice of $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ it will not be able to establish the safety of the command `delete x` in one of the programs (a) and (b).

Calcagno et al. [14] have recently proposed an analysis for resource invariant inference that makes decisions about ownership transfer by considering all critical regions in the program. Their analysis also relies on ad hoc design decisions, albeit different in nature from the ones we have to make here.

The conjunction rule is hard to keep sound in settings other than concurrency. For example, its soundness in the logic for information hiding in heap-manipulating programs [58] relies on the requirement of precision of module invariants. Similar issues come up in logics for concurrency that use relations instead of invariants to describe interference, such as RGSep [77]. We conjecture that the techniques used in this dissertation to establish the soundness of logics without conjunction rule and additional restrictions can be used to obtain similar results in other settings.

The use of abstract interpretation with state separation is not limited to the analysis of concurrent programs: for example, in Chapter 5 we show that it can be used to scale up interprocedural analyses. In fact, apart from the author's publication on the subject with Josh Berdine, Byron Cook, and Mooly Sagiv [34], the idea of using generic abstract domains with a $*$-like operation has been independently arrived at by Dino Distefano, Peter O'Hearn and Hongseok Yang and, separately, Noam Rinetzky. A similar framework has been also proposed by Retert and Boyland [65].

Chapter 4

Storable locks

Concurrent separation logic and the corresponding program analysis presented in Chapter 3 suffer from a common limitation: they assume a bounded number of non-aliased and pre-allocated locks and, hence, cannot be used to reason about concurrency primitives present in modern languages and libraries (e.g., POSIX threads [44]) that use unbounded numbers of *storable* locks dynamically allocated and destroyed in the heap.

Reasoning about storable locks is challenging. The issue here is not that of expressivity, but of modularity: storable locks can be handled by building a global invariant describing the shared memory as a whole, with all locks allocated in it. However, in this case the locality of reasoning is lost, which kicks back in global invariants containing lots of auxiliary state, proofs being extremely complex, and program analyses for discovery of global invariants being infeasible. Recent efforts towards making proofs in this style of reasoning modular [27, 77] use rely-guarantee reasoning to simplify the description of the global invariant and its possible changes (see Section 4.7 for a detailed comparison of such techniques with our work).

We want to preserve concurrent separation logic's local reasoning, even for programs that manipulate storable locks. To this end, in this chapter we propose a new logic (Section 4.1), based upon separation logic, and a corresponding analysis (Section 4.6) that treat storable locks along with the data structures they protect as resources, assigning invariants to them and managing their dynamic creation and destruction. The challenges of reasoning about storable locks were (quite emotionally) summarised by Bornat et al. [7]:

> ...the idea of semaphores in the heap makes theoreticians wince. The semaphore has to be available to a shared resource bundle:[1] that means a bundle will contain a bundle which contains resource, a notion which makes everybody's eyes water.

Less emotionally, storable locks are analogous to storable procedures in that, unless one is very careful, they can raise a form of Russell's paradox, circularity arising from what

[1]Here the term "resource bundle" is used to name what we, following O'Hearn's original paper [56], call "resource invariant".

Landin called knots in the store. Storable locks can do this by referring to themselves through their resource invariants, and here we address this foundational difficulty by cutting the knots in the store with an indirection.

Our approach to reasoning about storable locks is to represent a lock in the assertion language by a *handle* whose denotation cuts knots in the store. A handle certifies that a lock allocated at a certain address exists and gives a thread owning the handle a permission to (try to) acquire the lock. Using the mechanism of permissions [10, 7], the handle can be split among several threads that can then compete for the lock. Furthermore, a handle carries some information about the part of the program state protected by the lock (its resource invariant), which lets us mediate the interaction among threads, just as in the original concurrent separation logic. Handles for locks can be stored inside resource invariants, thereby permitting reasoning about the situation described in the quote above. In this way we extend the ability of concurrent separation logic to reason locally about programs that are consistent with the Ownership Hypothesis to the setting with storable locks. As we show in Section 4.2, the class of such programs contains programs with coarse-grained synchronisation and some, but not all, programs with fine-grained synchronisation, including examples that were posed as challenges in the literature. Due to relatively complicated semantic structures needed to interpret handles, in this chapter we interpret the assertion language of the logic with respect to a single model (instead of a class of models, as in Chapter 3), which we define in Section 4.3.

We prove the logic sound with the same method we used in Section 3.4 for proving the soundness of the logic for static locks (Section 4.5). In particular, the scheme of the corresponding thread-modular analysis is obtained directly from the thread-local interpretation used in the proof of the logic. However, in the case of storable locks even formulating the soundness statement is non-trivial as we have to take into account resource invariants for locks not mentioned directly in the local states of threads. As in Chapter 3, we prove the soundness of two variants of the logic: one that requires resource invariants of locks to be precise, but includes the conjunction rule, and one that omits the rule and the restriction.

4.1 A logic for storable locks

We extend the programming language of Section 2.1.3 with the following commands:

$$C \quad ::= \quad \ldots \mid \texttt{init}(E) \mid \texttt{finalise}(E) \mid \texttt{acquire}(E) \mid \texttt{release}(E)$$

where E ranges over expressions (Section 2.1.2). As before, we assume that each program consists of a parallel composition of several threads. Synchronisation is performed using locks, which are dynamically created and destroyed in the heap.

The $\texttt{acquire}(E)$ and $\texttt{release}(E)$ commands try to acquire, respectively, release the lock allocated at the address E. Locks have the semantics of binary semaphores, i.e., we allow a thread distinct from the one that acquired a lock to release it. The technical development in this chapter can be adjusted to the case when locks have the semantics of mutexes, i.e., when a lock can only be released by the thread that acquired it ([32] provides such treatment). The `init`(E) command converts a location allocated at the address E to a lock. After the completion of `init`(E), the lock is held. The `finalise`(E) command converts the lock into an ordinary heap cell containing an unspecified value provided that the lock at the address E is held.

As in concurrent separation logic, with each lock we associate a resource invariant—a formula that describes the part of the heap protected by the lock. To deal with unbounded numbers of locks, we assume that each lock has a *sort* that determines its invariant. Formally, we assume a fixed set $\mathcal{A}$ of function symbols with positive arities representing lock sorts, and with each $A \in \mathcal{A}$ of arity k and parameters $L, \vec{X}$ we associate a formula $I_A(L, \vec{X})$ containing k free logical variables specified as parameters—the resource invariant for the sort A. The meaning of the first parameter is fixed as the address at which the lock is allocated. Other parameters can have arbitrary meaning. We denote with $\mathcal{A}_k$ the set of lock sorts of arity k.

We extend the assertion language of Section 2.1.1 with two extra forms:

$$P \quad ::= \quad \ldots \mid \pi\mathsf{Lock}_A(E, \vec{F}) \mid \mathsf{Hold}_A(E, \vec{F})$$

An assertion of the form $\mathsf{Lock}_A(E, \vec{F})$, where $A \in \mathcal{A}$, is a *handle* for the lock of the sort A allocated at the address E. It can be viewed as an existential permission for the lock: a thread having $\mathsf{Lock}_A(E, \vec{F})$ knows that the heap cell at the address E is allocated and is a lock, and can try to acquire it. The assertion $\mathsf{Lock}_A(E, \vec{F})$ does not give permissions for reading from or writing to the cell at the address E. Moreover, it does not ensure that the part of the heap protected by the lock satisfies the resource invariant until the thread successfully acquires the lock. We allow using $\mathsf{Lock}_A(E, \vec{F})$ with fractional permissions (Section 2.1.1), writing $\pi\mathsf{Lock}_A(E, \vec{F})$. The intuition behind the permissions is that a handle for a lock with the full permission 1 can be split among several threads, thereby allowing them to compete for the lock. A thread having a permission for the handle less than 1 can acquire the lock; a thread having the full permission can in addition finalise the lock. We abbreviate $1\mathsf{Lock}_A(E, \vec{F})$ to $\mathsf{Lock}_A(E, \vec{F})$. Assertions in the code of threads can also use a special form $\mathsf{Hold}_A(E, \vec{F})$ to represent the fact that the lock at the address E is held by the thread in the surrounding code of the assertion. The assertion $\mathsf{Hold}_A(E, \vec{F})$ also ensures that the cell at the address E is allocated and is a lock of the sort A with the parameters $\vec{F}$.

Our logic extends sequential separation logic (Section 2.1.3) with the axioms for lock-manipulating commands shown in Figure 4.1. The judgements of the logic are of the form $I \vdash \{P\}\ C\ \{Q\}$, where I maps lock sorts from the set $\mathcal{A}$ to the corresponding resource invariants. Since we treat locks as binary semaphores, in our logic resource invariants can contain Hold-facts, which can then be transferred between threads. As in Chapter 3, we consider two variants of the logic: one that includes the conjunction rule, but requires

$$\frac{(O \Vdash E \mapsto \_) \Rightarrow \vec{F} = \vec{F}}{I \vdash \{O \Vdash E \mapsto \_\}\ \texttt{init}(E)\ \{O \Vdash \mathsf{Lock}_A(E, \vec{F}) * \mathsf{Hold}_A(E, \vec{F})\}}\ \textsc{Init}$$

$$\frac{(O \Vdash E \mapsto \_) \Rightarrow \vec{F} = \vec{F}}{I \vdash \{O \Vdash E \mapsto \_\}\ \texttt{init}_{A,\vec{F}}(E)\ \{O \Vdash \mathsf{Lock}_A(E, \vec{F}) * \mathsf{Hold}_A(E, \vec{F})\}}\ \textsc{Init}'$$

$$\frac{}{I \vdash \{O \Vdash \mathsf{Lock}_A(E, \vec{F}) * \mathsf{Hold}_A(E, \vec{F})\}\ \texttt{finalise}(E)\ \{O \Vdash E \mapsto \_\}}\ \textsc{Finalise}$$

$$\frac{}{\begin{array}{l} I \vdash \{(O \Vdash \pi\mathsf{Lock}_A(L, \vec{X})) \wedge L = E\} \\ \qquad \texttt{acquire}(E) \\ \qquad \{(O \Vdash \pi\mathsf{Lock}_A(L, \vec{X}) * \mathsf{Hold}_A(L, \vec{X})) * I_A(L, \vec{X})\} \end{array}}\ \textsc{Acquire}$$

$$\frac{}{I \vdash \{((O \Vdash \mathsf{Hold}_A(L, \vec{X})) * I_A(L, \vec{X})) \wedge L = E\}\ \texttt{release}(E)\ \{O \Vdash \mathsf{emp}_{\mathbf{h}}\}}\ \textsc{Release}$$

Figure 4.1: Proof rules for lock-manipulating commands

resource invariants to be precise (with respect to the model we define in Section 4.3), and one that drops the rule and the restriction.

Initialising a lock (Init) converts a cell in the heap at the address E to a lock. Upon the completion of $\texttt{init}(E)$ the thread that executed it gets both the ownership (with the full permission) of the handle $\mathsf{Lock}_A(E, \vec{F})$ for the lock and the knowledge that the lock is held, represented by $\mathsf{Hold}_A(E, \vec{F})$. For the `init` command to be safe, the stack must contain the variables mentioned in E and $\vec{F}$, hence, the premiss $(O \Vdash E \mapsto \_) \Rightarrow \vec{F} = \vec{F}$ additionally requires that variables be contained in O. In the variant of the logic with the conjunction rule, the sort of the lock that is being created and its parameters have to be chosen consistently for each `init` command. We can enforce this by annotating each `init` command with the sort A and the parameters $\vec{F}$, the latter defined by arbitrary expressions over program variables[2] (Init′).

Finalising a lock results in it being converted into an ordinary cell. To finalise a lock (Finalise) a thread has to have the full permission for the handle $\mathsf{Lock}_A(E, \vec{F})$ associated with the lock. Additionally, the lock has to be held, i.e., $\mathsf{Hold}_A(E, \vec{F})$ has to be in the local state of the thread.

A thread can acquire a lock if it has a permission for the handle of the lock. As in concurrent separation logic, acquiring a lock (Acquire) results in the resource invariant of the lock (with appropriately instantiated parameters) being $*$-conjoined to the local state of the thread. The thread also obtains the corresponding Hold fact, which guarantees that the lock is held. Acquiring the same lock twice leads to a deadlock, which is enforced by

[2] This can be generalised to the case when $\vec{F}$ depend on the heap.

$\mathsf{Hold}_A(E, \vec{F}) * \mathsf{Hold}_A(E, \vec{F})$ being inconsistent in our model of the assertion language (see Section 4.3). Conversely, a thread can release a lock (Release) only if the lock is held, i.e., the corresponding Hold fact is present in the local state of the thread. Note that since Hold ensures the existence of the lock, we do not require a Lock fact in the precondition of Release. Upon releasing the lock the thread gives up both this knowledge and the ownership of the resource invariant associated with the lock. The fact that resource invariants can claim ownership of program variables complicates the axioms Acquire and Release. For example, in the postcondition of Acquire we cannot put $I_A(L, \vec{X})$ inside the expression after $\Vdash$ as it may claim ownership of variables not mentioned in O. This requires us to use a logical variable L in places where the expression E would have been expected.

4.2 Examples of reasoning

We first show (in Example 1 below) that straightforward application of rules for lock-manipulating commands allows us to handle programs in which locks protect parts of the heap without other locks allocated in them. We then present two more involved examples of using the logic, which demonstrate how extending the logic with storable locks has enabled reasoning more locally than was previously possible in some interesting cases (Examples 2 and 3). As in Section 3.1, we use a C-like language for our examples. We defer the definition of the formal model of the assertion language to Section 4.3, appealing to the informal explanations given in the previous section.

Example 1: A simple situation. Figure 4.2 shows a proof outline for a program with a common pattern: a lock-field in a structure protecting another field in the same structure. We use a lock sort R with the invariant

$$I_R(L) = \mathsf{emp_s} \wedge L.\mathtt{Data} \mapsto \_.$$

The proof outline shows how the "life cycle" of a lock is handled in our proof system: creating a cell, converting it to a lock, acquiring and releasing the lock, converting it to an ordinary cell, and disposing the cell. For simplicity we consider a program with only one thread. □

Example 2: "Last one disposes". This example was posed as a challenge for local reasoning by Bornat et al. [7]. The program in Figure 4.3 represents a piece of multicasting code: a single packet `p` (of type `PACKET`) with `Data` inside the packet is distributed to M `threads` at once. For efficiency reasons instead of copying the packet, it is shared among threads. A `Count` of access permissions protected by `Lock` is used to determine when everybody has finished and the packet can be disposed of. The program consists

```
struct RECORD {
  LOCK Lock;
  int Data;
};

main() {
  RECORD *x;
  $\{\mathtt{x} \Vdash \mathsf{emp}_{\mathbf{h}}\}$
  x = new RECORD;
  $\{\mathtt{x} \Vdash \mathtt{x} \mapsto \_ * \mathtt{x.Data} \mapsto \_\}$
  init$_R$(x);
  $\{\mathtt{x} \Vdash \mathtt{x.Data} \mapsto \_ * \mathsf{Lock}_R(\mathtt{x}) * \mathsf{Hold}_R(\mathtt{x})\}$
  x->Data = 0;
  $\{\mathtt{x} \Vdash \mathtt{x.Data} \mapsto 0 * \mathsf{Lock}_R(\mathtt{x}) * \mathsf{Hold}_R(\mathtt{x})\}$
  release(x);
  $\{\mathtt{x} \Vdash \mathsf{Lock}_R(\mathtt{x})\}$
  // ...
  $\{\mathtt{x} \Vdash \mathsf{Lock}_R(\mathtt{x})\}$
  acquire(x);
  $\{\mathtt{x} \Vdash \mathtt{x.Data} \mapsto \_ * \mathsf{Lock}_R(\mathtt{x}) * \mathsf{Hold}_R(\mathtt{x})\}$
  x->Data++;
  $\{\mathtt{x} \Vdash \mathtt{x.Data} \mapsto \_ * \mathsf{Lock}_R(\mathtt{x}) * \mathsf{Hold}_R(\mathtt{x})\}$
  release(x);
  $\{\mathtt{x} \Vdash \mathsf{Lock}_R(\mathtt{x})\}$
  // ...
  $\{\mathtt{x} \Vdash \mathsf{Lock}_R(\mathtt{x})\}$
  acquire(x);
  $\{\mathtt{x} \Vdash \mathtt{x.Data} \mapsto \_ * \mathsf{Lock}_R(\mathtt{x}) * \mathsf{Hold}_R(\mathtt{x})\}$
  finalise(x);
  $\{\mathtt{x} \Vdash \mathtt{x} \mapsto \_ * \mathtt{x.Data} \mapsto \_\}$
  delete x;
  $\{\mathtt{x} \Vdash \mathsf{emp}_{\mathbf{h}}\}$
}
```

Figure 4.2: A simple example of reasoning in the logic

```
struct PACKET { LOCK Lock; int Count; DATA Data; };
PACKET *p;

thread() {
  {$(1/M)\mathsf{p} \Vdash (1/M)\mathsf{Lock}_P(\mathsf{p}, M)$}
  acquire(p);
  {$(1/M)\mathsf{p} \Vdash \exists X.\, 0 \le X < M \wedge \mathsf{p.Count}\mapsto X * \mathsf{p.Data}\mapsto\_ * ((X+1)/M)\mathsf{Lock}_P(\mathsf{p}, M) *$
   $\mathsf{Hold}_P(\mathsf{p}, M)$}
  // ...Process data...
  p->Count++;
  {$(1/M)\mathsf{p} \Vdash \exists X.\, 1 \le X \le M \wedge \mathsf{p.Count}\mapsto X * \mathsf{p.Data}\mapsto\_ * (X/M)\mathsf{Lock}_P(\mathsf{p}, M) * \mathsf{Hold}_P(\mathsf{p}, M)$}
  if (p->Count == M) {
    {$(1/M)\mathsf{p} \Vdash \mathsf{p.Count}\mapsto M * \mathsf{p.Data}\mapsto\_ * \mathsf{Lock}_P(\mathsf{p}, M) * \mathsf{Hold}_P(\mathsf{p}, M)$}
    // ...Finalise data...
    finalise(p);
    {$(1/M)\mathsf{p} \Vdash \mathsf{p.Count}\mapsto M * \mathsf{p.Data}\mapsto\_ * \mathsf{p}\mapsto\_$}
    delete(p);
  } else {
    {$(1/M)\mathsf{p} \Vdash \exists X.\, 1 \le X < M \wedge \mathsf{p.Count}\mapsto X * \mathsf{p.Data}\mapsto\_ * (X/M)\mathsf{Lock}_P(\mathsf{p}, M) *$
     $\mathsf{Hold}_P(\mathsf{p}, M)$}
    release(p);
  }
  {$(1/M)\mathsf{p} \Vdash \mathsf{emp}_{\mathbf{h}}$}
}

initialise() {
  {$\mathsf{p} \Vdash \mathsf{emp}_{\mathbf{h}}$}
  p = new PACKET;
  {$\mathsf{p} \Vdash \mathsf{p}\mapsto\_ * \mathsf{p.Count}\mapsto\_ * \mathsf{p.Data}\mapsto\_$}
  p->Count = 0;
  {$\mathsf{p} \Vdash \mathsf{p}\mapsto\_ * \mathsf{p.Count}\mapsto 0 * \mathsf{p.Data}\mapsto\_$}
  init$_P$(p);
  // ...Initialise data...
  {$\mathsf{p} \Vdash \mathsf{p.Count}\mapsto 0 * \mathsf{p.Data}\mapsto\_ * \mathsf{Lock}_P(\mathsf{p}, M) * \mathsf{Hold}_P(\mathsf{p}, M)$}
  release(p);
  {$\mathsf{p} \Vdash \mathsf{Lock}_P(\mathsf{p}, M)$}
}
```

Figure 4.3: Proof outline for the “Last one disposes” program

of a top-level parallel composition of M calls to the procedure `thread`. Here M is a constant assumed to be greater than 0. For completeness, we also provide the procedure `initialise` that can be used to initialise the packet and thereby establish the precondition of the program.

To prove the program correct, the resource invariant for the lock at the address `p` has to contain a partial permission for the handle of *the same lock*. This is formally represented by a lock sort P with the resource invariant

$$I_P(L, M) = \mathsf{emp_s} \land \exists X.\, X < M \land L.\mathtt{Count} \mapsto X * L.\mathtt{Data} \mapsto \_ * \\ ((X = 0 \land \mathsf{emp_h}) \lor (X \geq 1 \land (X/M)\mathsf{Lock}_P(L, M))).$$

Initially the resource invariant contains no permissions of this kind and the handle $\mathsf{Lock}_P(\mathtt{p}, M)$ for the lock is split among M threads (hence, the precondition of each thread is $(1/M)\mathtt{p} \Vdash (1/M)\mathsf{Lock}_P(\mathtt{p}, M)$). Each thread uses the handle to acquire the lock and process the packet. When a thread finishes processing and releases the lock, it transfers the permission for the handle it owned to the resource invariant of the lock. The last thread to process the packet can then get the full permission for the lock by combining the permission in the invariant with its own one and can therefore dispose of the packet. $\square$

Example 3: Lock coupling list. We next consider a fine-grained implementation of a singly-linked list with concurrent access, whose nodes store integer keys. The program (Figures 4.4 and 4.5) consists of M operations `add` and `remove` running in parallel. The operations add and remove an element with the given key to or from the list. Traversing the list uses lock coupling: the lock on one node is not released until the next node is locked. For the purposes of this example, we assume that the set of values stored in ordinary memory cells Values $= \{\mathtt{-INF}, \ldots, 0, \ldots, \mathtt{INF}\}$, so that `INF` is the biggest representable integer. The list is sorted and its first and last nodes are sentinel nodes that have values `-INF`, respectively, `INF`. It is initialised by the code in the procedure `initialise`. We only provide a proof outline for the procedure `locate` (Figure 4.4), which is invoked by other procedures to traverse the list. We use lock sorts H (for the head node) and N (for all other nodes) with the following invariants:

$$\begin{aligned} I_H(L) &= \mathsf{emp_s} \land \exists X, V'.\, L.\mathtt{Val} \mapsto \mathtt{-INF} * L.\mathtt{Next} \mapsto X * \mathsf{Lock}_N(X, V') \land \mathtt{-INF} < V'; \\ I_N(L, V) &= \mathsf{emp_s} \land ((L.\mathtt{Val} \mapsto V * L.\mathtt{Next} \mapsto \mathtt{NULL} \land V = \mathtt{INF}) \lor \\ &\quad (\exists X, V'.\, L.\mathtt{Val} \mapsto V * L.\mathtt{Next} \mapsto X * \mathsf{Lock}_N(X, V') \land V < V')). \end{aligned}$$

In this example, the resource invariant for the lock protecting a node in the list holds a handle for the lock protecting the next node in the list. The full permission for $\mathsf{Lock}_N(X, V')$ in the invariants above essentially means that the only way a thread can lock a node is by first locking its predecessor: here the invariant enforces a particular locking policy. $\square$

```
locate(int e) {
  NODE *prev, *curr;
  $\{O \Vdash \texttt{-INF} < \mathsf{e} \wedge (1/M)\mathsf{Lock}_H(\texttt{head})\}$
  prev = head;
  $\{O \Vdash \texttt{-INF} < \mathsf{e} \wedge \texttt{prev} = \texttt{head} \wedge (1/M)\mathsf{Lock}_H(\texttt{head})\}$
  acquire(prev);
  $\{O \Vdash \exists V'.\, \texttt{-INF} < \mathsf{e} \wedge \texttt{-INF} < V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_H(\texttt{prev}) *$
   $\exists X.\, \texttt{prev.}\mathsf{Val}{\mapsto}\texttt{-INF} * \texttt{prev.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V')\}$
  curr = prev->Next;
  $\{O \Vdash \exists V'.\, \texttt{-INF} < \mathsf{e} \wedge \texttt{-INF} < V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_H(\texttt{prev}) *$
   $\texttt{prev.}\mathsf{Val}{\mapsto}\texttt{-INF} * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} * \mathsf{Lock}_N(\texttt{curr}, V')\}$
  acquire(curr);
  $\{O \Vdash \exists V'.\, \texttt{-INF} < \mathsf{e} \wedge \texttt{-INF} < V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Lock}_N(\texttt{curr}, V') * \mathsf{Hold}_H(\texttt{prev}) *$
   $\mathsf{Hold}_N(\texttt{curr}, V') * \texttt{prev.}\mathsf{Val}{\mapsto}\texttt{-INF} * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} * \texttt{curr.}\mathsf{Val}{\mapsto}V' *$
   $((\texttt{curr.}\mathsf{Next}{\mapsto}\texttt{NULL} \wedge V' = \texttt{INF}) \vee (\exists X, V''.\, \texttt{curr.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V'') \wedge V' < V''))\}$
  while (curr->Val < e) {
    $\{O \Vdash \exists V, V'.\, V' < \mathsf{e} \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Lock}_N(\texttt{curr}, V') * \mathsf{Hold}_N(\texttt{curr}, V') *$
     $(\mathsf{Hold}_H(\texttt{prev}) \wedge V = \texttt{-INF} \vee \mathsf{Hold}_N(\texttt{prev}, V)) * \texttt{prev.}\mathsf{Val}{\mapsto}V * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} *$
     $\exists X, V''.\, \texttt{curr.}\mathsf{Val}{\mapsto}V' * \texttt{curr.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V'') \wedge V < V' < V''\}$
    release(prev);
    $\{O \Vdash \exists X, V', V''.\, V' < \mathsf{e} \wedge V' < V'' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_N(\texttt{curr}, V') *$
     $\texttt{curr.}\mathsf{Val}{\mapsto}V' * \texttt{curr.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V'')\}$
    prev = curr;
    curr = curr->Next;
    $\{O \Vdash \exists V, V'.\, V < \mathsf{e} \wedge V < V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_N(\texttt{prev}, V) *$
     $\texttt{prev.}\mathsf{Val}{\mapsto}V * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} * \mathsf{Lock}_N(\texttt{curr}, V')\}$
    acquire(curr);
    $\{O \Vdash \exists V, V'.\, V < \mathsf{e} \wedge V < V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_N(\texttt{prev}, V) *$
     $\mathsf{Hold}_N(\texttt{curr}, V') * \mathsf{Lock}_N(\texttt{curr}, V') * \texttt{prev.}\mathsf{Val}{\mapsto}V * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} * \texttt{curr.}\mathsf{Val}{\mapsto}V' *$
     $((V' = \texttt{INF} \wedge \texttt{curr.}\mathsf{Next}{\mapsto}\texttt{NULL}) \vee \exists X, V''.\, \texttt{curr.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V'') \wedge V' < V'')\}$
  }
  $\{O \Vdash \exists V, V'.\, V < \mathsf{e} \le V' \wedge (1/M)\mathsf{Lock}_H(\texttt{head}) * \mathsf{Hold}_N(\texttt{prev}, V) * \mathsf{Hold}_N(\texttt{curr}, V') *$
   $\mathsf{Lock}_N(\texttt{curr}, V') * \texttt{prev.}\mathsf{Val}{\mapsto}V * \texttt{prev.}\mathsf{Next}{\mapsto}\texttt{curr} * \texttt{curr.}\mathsf{Val}{\mapsto}V' *$
   $((V' = \texttt{INF} \wedge \texttt{curr.}\mathsf{Next}{\mapsto}\texttt{NULL}) \vee \exists X, V''.\, \texttt{curr.}\mathsf{Next}{\mapsto}X * \mathsf{Lock}_N(X, V'') \wedge V' < V'')\}$
  return (prev, curr);
}
```

Figure 4.4: Proof outline for a part of the lock coupling list program. Here O is `e`, `prev`, `curr`, $(1/M)$`head`.

```
struct NODE {
  LOCK Lock;
  int Val;
  NODE *Next;
};
NODE *head;

initialise() {
  NODE *last;
  last = new NODE;
  last->Val = INF;
  last->Next = NULL;
  init$_{N,\mathtt{INF}}$(last);
  release(last);
  head = new NODE;
  head->Val = -INF;
  head->Next = last;
  init$_H$(head);
  release(head);
}
```

```
add(int e) {
  NODE *n1, *n2, *n3;
  NODE *result;
  (n1, n3) = locate(e);
  if (n3->Val != e) {
    n2 = new NODE;
    n2->Val = e;
    n2->Next = n3;
    init$_{N,\mathtt{e}}$(n2);
    release(n2);
    n1->Next = n2;
    result = true;
  } else {
    result = false;
  }
  release(n1);
  release(n3);
  return result;
}
```

```
remove(int e) {
  NODE *n1, *n2, *n3;
  NODE *result;
  (n1, n2) = locate(e);
  if (n2->Val == e) {
    n3 = n2->Next;
    n1->Next = n3;
    finalise(n2);
    delete n2;
    result = true;
  } else {
    release(n2);
    result = false;
  }
  release(n1);
  return result;
}
```

Figure 4.5: Lock coupling list program. The procedure `locate` is shown in Figure 4.4.

We were able to present modular proofs for the programs above because they satisfied the Ownership Hypothesis: in each case we could associate with every lock a part of the heap such that a thread accessed the part only when it held the lock, that is, the lock owned the part of the heap. We note that we would not be able to give modular proofs to programs that do not obey this policy, for instance, to optimistic list [40, Section 9.6]—another fine-grained implementation of the list from Example 3 in which the procedure `locate` first traverses the list without taking any locks and then validates the result by locking two candidate nodes and re-traversing the list to check that they are still present and adjacent in the list. We discuss methods for reasoning about such programs in Section 4.7.

4.3 Model of the assertion language

We modify the model States of separation logic's assertions presented in Section 2.1.1 as shown in Figure 4.6. As before, assertion language formulae denote sets of triples of a stack, a heap, and an interpretation. However, in contrast to the domain of Section 2.1.1, here cells in the heap can be of two types: ordinary cells (**Cell**) and locks (**Lock**). A lock is associated with a sort, a list of parameters of its resource invariant (corresponding to

$$
\begin{aligned}
&\text{Values} = \{\dots, -1, 0, 1, \dots\}\\
&\text{Perms} = (0, 1]\\
&\text{Locs} = \{1, 2, \dots\}\\
&\text{Vars} = \{\mathtt{x}, \mathtt{y}, \dots\}\\
&\text{LVars} = \{X, Y, \dots\}\\
&\text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms})\\
&\text{LockPerms} = [0, 1]\\
&\text{LockVals} = \{\mathsf{L}, \mathsf{F}, \mathsf{U}\}\\
&\text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \Bigg(\mathbf{Cell}(\text{Values}) \cup \bigcup_{k\geq 0} (\mathbf{Lock}(\mathcal{A}_{k+1} \times \text{Values}^k \times \text{LockVals} \times \text{LockPerms}) \setminus \\
&\qquad\qquad\qquad\qquad \mathbf{Lock}(\mathcal{A}_{k+1} \times \text{Values}^k \times \{\mathsf{U}\} \times \{0\}))\Bigg)\\
&\text{Ints} = \text{LVars} \rightarrow \text{Values}\\
&\text{States} = \text{Stacks} \times \text{Heaps} \times \text{Ints}
\end{aligned}
$$

Figure 4.6: Model of the assertion language

the arity of the sort), a value, and a permission from $[0, 1]$. The permission 0 is used to represent the existential permission for a lock that is carried by $\mathsf{Hold}_A(E, \vec{F})$. The values of locks are interpreted as follows: L means that the lock is held, F that the lock is free, and U that the status of the lock is undefined. The value U is not encountered in the states obtained in the operational semantics, but is used for interpreting formulae representing parts of complete states. Additionally, the semantics of formulae and commands never encounter locks of the form $\mathbf{Lock}(A, \vec{w}, \mathsf{U}, 0)$ for any A and $\vec{w}$, hence, the definition of Heaps removes them to make the $*$ operation on states cancellative.

One may be tempted to allow locks to refer to resource invariants directly in the model, i.e., to define Heaps as

$$
\begin{aligned}
\text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \Bigg(&\mathbf{Cell}(\text{Values}) \cup \bigcup_{k\geq 0} (\mathbf{Lock}(\mathcal{P}(\text{States}) \times \text{LockVals} \times \text{LockPerms}) \setminus \\
&\mathbf{Lock}(\mathcal{P}(\text{States}) \times \{\mathsf{U}\} \times \{0\}))\Bigg)
\end{aligned}
\tag{4.1}
$$

(of course, this model would require a richer assertion language). Unfortunately, such a recursive definition leads to the set-theoretic paradox mentioned at the beginning of this chapter: the equations defining the model have no solutions in sets because the cardinality of $\mathcal{P}(\text{States})$ is strictly bigger than the cardinality of States. In the model of Figure 4.6, Heaps is not defined recursively, but instead uses an indirection through $\mathcal{A}$, whose elements are associated with resource invariants, and hence indirectly to Heaps. It is this indirection that deals with the foundational circularity issue raised by locks which may refer to themselves and makes the definition of the model well-formed. We discuss the pros and cons of this solution in Section 4.7.

We turn the set States into a separation algebra by defining the $*$ operation as follows. We first define $*$ on values of locks in the following way:

$$\mathsf{U} * \mathsf{U} = \mathsf{U},\ a * \mathsf{U} = \mathsf{U} * a = a,\ a * b \text{ is undefined,} \tag{4.2}$$

where $a, b \in \{\mathsf{L}, \mathsf{F}\}$. Note that $\mathsf{L} * \mathsf{L}$ is undefined as it arises in the cases when a thread tries to acquire a lock twice (recall that we specify that a thread deadlocks in this case). For $s_1, s_2 \in$ Stacks let

$$\begin{aligned} s_1 \natural s_2 \Leftrightarrow (\forall \mathtt{x}.\, s_1(\mathtt{x})\!\downarrow \wedge s_2(\mathtt{x})\!\downarrow \Rightarrow \\ (\exists u, \pi_1, \pi_2.\, s_1(\mathtt{x}) = (u, \pi_1) \wedge s_2(\mathtt{x}) = (u, \pi_2) \wedge \pi_1 + \pi_2 \leq 1)). \end{aligned}$$

If $s_1 \natural s_2$, then we define

$$\begin{aligned} s_1 * s_2 = \{(\mathtt{x}, (u, \pi)) \mid (s_1(\mathtt{x}) = (u, \pi) \wedge s_2(\mathtt{x})\!\uparrow) \vee (s_2(\mathtt{x}) = (u, \pi) \wedge s_1(\mathtt{x})\!\uparrow) \vee \\ (s_1(\mathtt{x}) = (u, \pi_1) \wedge s_2(\mathtt{x}) = (u, \pi_2) \wedge \pi = \pi_1 + \pi_2)\}; \end{aligned}$$

otherwise $s_1 * s_2$ is undefined. For $h_1, h_2 \in$ Heaps let

$$\begin{aligned} h_1 \natural h_2 \Leftrightarrow (\forall u.\, h_1(u)\!\downarrow \wedge h_2(u)\!\downarrow \Rightarrow (\exists A, b_1, b_2, \pi_1, \pi_2, \vec{w}.\, (b_1 * b_2)\!\downarrow \wedge \pi_1 + \pi_2 \leq 1 \wedge \\ h_1(u) = \mathbf{Lock}(A, \vec{w}, b_1, \pi_1) \wedge h_2(u) = \mathbf{Lock}(A, \vec{w}, b_2, \pi_2))). \end{aligned}$$

If $h_1 \natural h_2$, then we define

$$\begin{gathered} h_1 * h_2 = \{(u, \mathbf{Cell}(b)) \mid h_1(u) = \mathbf{Cell}(b) \vee h_2(u) = \mathbf{Cell}(b)\} \cup \{(u, \mathbf{Lock}(A, \vec{w}, b, \pi)) \mid \\ (h_1(u) = \mathbf{Lock}(A, \vec{w}, b, \pi) \wedge h_2(u)\!\uparrow) \vee (h_2(u) = \mathbf{Lock}(A, \vec{w}, b, \pi) \wedge h_1(u)\!\uparrow) \\ \vee (h_1(u) = \mathbf{Lock}(A, \vec{w}, b_1, \pi_1) \wedge h_2(u) = \mathbf{Lock}(A, \vec{w}, b_2, \pi_2) \wedge \pi = \pi_1 + \pi_2 \wedge b = b_1 * b_2)\}; \end{gathered}$$

otherwise $h_1 * h_2$ is undefined. For $(s_1, h_1, \mathbf{i}_1) \in$ States and $(s_2, h_2, \mathbf{i}_2) \in$ States we then let

$$(s_1, h_1, \mathbf{i}_1) * (s_2, h_2, \mathbf{i}_2) = (s_1 * s_2, h_1 * h_2, \mathbf{i}_1 * \mathbf{i}_2),$$

where $*$ on interpretations is defined in Section 2.1.1.

Since, in our logic, assertions cannot state the fact that a particular lock is free, they are actually interpreted over a set of *local states* $\text{States}_\mathbf{l} \subseteq$ States that is defined as in Figure 4.6, but with LockVals $= \{\mathsf{L}, \mathsf{U}\}$. States containing locks with the value F are used to formulate the soundness theorem for the logic. The satisfaction relation for the assertion language formulae is defined by extending the one in Figure 2.2 with clauses for the new forms:

$$\begin{aligned} (s, h, \mathbf{i}) \models \pi\mathsf{Lock}_A(E, \vec{F}) \ &\Leftrightarrow\ [\![E]\!]_{s,\mathbf{i}}\!\downarrow \wedge [\![\vec{F}]\!]_{s,\mathbf{i}}\!\downarrow \wedge [\![\pi]\!]_{s,\mathbf{i}}\!\downarrow \wedge \\ &\qquad h = [[\![E]\!]_{s,\mathbf{i}} : \mathbf{Lock}(A, [\![\vec{F}]\!]_{s,\mathbf{i}}, \mathsf{U}, [\![\pi]\!]_{s,\mathbf{i}})] \wedge 0 < [\![\pi]\!]_{s,\mathbf{i}} \leq 1 \\ (s, h, \mathbf{i}) \models \mathsf{Hold}_A(E, \vec{F}) \ &\Leftrightarrow\ [\![E]\!]_{s,\mathbf{i}}\!\downarrow \wedge [\![\vec{F}]\!]_{s,\mathbf{i}}\!\downarrow \wedge h = [[\![E]\!]_{s,\mathbf{i}} : \mathbf{Lock}(A, [\![\vec{F}]\!]_{s,\mathbf{i}}, \mathsf{L}, 0)] \end{aligned}$$

Let $D = \mathcal{P}(\text{States})^\top$ and $D_\mathbf{l} = \mathcal{P}(\text{States}_\mathbf{l})^\top$ be the separation domains constructed out of the separation algebras States and $\text{States}_\mathbf{l}$, respectively.

We say that $p \in \mathcal{P}(\text{States})$ depends only on logical variables $\vec{Y}$, if for any $(s, h, \mathbf{i}) \in p$, $u \in \text{Values}$, and $Z \notin \vec{Y}$ we have $(s, h, \mathbf{i}[Z : u]) \in p$. Let $D_\mathbf{l}[\vec{Y}]$ be the set of elements of $D_\mathbf{l}\backslash\{\top\}$ that depend only on $\vec{Y}$. We denote with InvMaps the set of *semantic resource invariant mappings*—functions $\mathcal{I} : \mathcal{A} \to D_\mathbf{l}\backslash\{\top\}$ such that for any $A \in \mathcal{A}$ we have $\mathcal{I}_A \in D_\mathbf{l}[\vec{Y}]$, where $\vec{Y}$ are the parameters of A (and the free variables of the corresponding resource invariant). The denotation of a resource invariant mapping I is a semantic resource invariant mapping $[\![I]\!] \in \text{InvMaps}$ that for every lock sort $A \in \mathcal{A}$ gives the denotation $[\![I]\!]_A$ of the corresponding resource invariant. For $\mathcal{I} \in \text{InvMaps}$ we let

$$\mathcal{I}^\mathsf{L}_A(u, \vec{w}) = \{(s, h, \mathbf{i}) \mid (s, h, \mathbf{i}[L : u][\vec{X} : \vec{w}]) \in \mathcal{I}_A * (\{[\,]\}\times\{[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 0)]\}\times\text{Ints})\};$$
$$\mathcal{I}^\mathsf{F}_A(u, \vec{w}) = \{(s, h, \mathbf{i}) \mid (s, h, \mathbf{i}[L : u][\vec{X} : \vec{w}]) \in \mathcal{I}_A * (\{[\,]\}\times\{[u : \mathbf{Lock}(A, \vec{w}, \mathsf{F}, 0)]\}\times\text{Ints})\},$$

where $L, \vec{X}$ are the parameters of A.

4.4 Semantics

Throughout the rest of this chapter we fix a program $S = C_1 \parallel \ldots \parallel C_n$. As the model of program states in the operational semantics we take the set States from Figure 4.6, but with

$$\text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \left(\mathbf{Cell}(\text{Values}) \cup \bigcup_{k\geq 0}(\mathbf{Lock}(\text{LockVals} \times \text{LockPerms})\backslash \right.$$
$$\left. \mathbf{Lock}(\{\mathsf{U}\} \times \{0\}))\right).$$

We call states defined in this way *concrete* and denote the set of them with $\text{States}_\mathbf{c}$. Note that states in $\text{States}_\mathbf{c}$ do not contain lock sorts and lock parameters: these are introduced by the logic and are not present in the states of the operational semantics we define here. However, as in the rest of this dissertation, we leave permissions and interpretations of logical variables in the states. As usual, we also consider the domain $D_\mathbf{c} = \mathcal{P}(\text{States}_\mathbf{c})^\top$ constructed out of the set $\text{States}_\mathbf{c}$. Let $\beta : \text{States} \to \text{States}_\mathbf{c}$ be the function that erases the information about lock sorts and lock parameters from concrete states. We lift it to D pointwise.

To define the semantics of primitive sequential commands, we assume local functions $f_C : \text{States} \to D$ for $C \in \mathsf{Seq}$ such that

$$\forall \xi_1, \xi_2 \in \text{States}.\, \beta(\xi_1) = \beta(\xi_2) \Rightarrow \beta(f_C(\xi_1)) = \beta(f_C(\xi_2)). \qquad (4.3)$$

Intuitively, (4.3) states that the functions f_C are insensitive to lock sorts and lock parameters recorded in the states from States: running a command C on two states with

$$\begin{array}{lll}
\texttt{init}(E), (s, h[\llbracket E \rrbracket_s : \mathbf{Cell}(u)], \mathbf{i}) & \rightsquigarrow & (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{L}, 1)], \mathbf{i}) \\
\texttt{finalise}(E), (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{L}, 1)], \mathbf{i}) & \rightsquigarrow & (s, h[\llbracket E \rrbracket_s : \mathbf{Cell}(u)], \mathbf{i}) \\
\texttt{acquire}(E), (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{F}, \pi)], \mathbf{i}) & \rightsquigarrow & (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{L}, \pi)], \mathbf{i}) \\
\texttt{acquire}(E), (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{L}, \pi)], \mathbf{i}) & \not\rightsquigarrow & \\
\texttt{release}(E), (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{L}, \pi)], \mathbf{i}) & \rightsquigarrow & (s, h[\llbracket E \rrbracket_s : \mathbf{Lock}(\mathsf{F}, \pi)], \mathbf{i}) \\
C, (s, h, \mathbf{i}) & \rightsquigarrow & \top, \text{ otherwise}
\end{array}$$

Figure 4.7: Transition relation on $\text{States}_\mathbf{c}$ for lock-manipulating commands. $\top$ indicates that the command faults. $\not\rightsquigarrow$ is used to denote that the command does not fault, but gets stuck.

different instrumentation and erasing the instrumentation in the results yields identical states. The Prim axiom of the logic is formulated using the transformers f_C. When the set of primitive sequential commands $\mathsf{Seq} = \mathsf{SeqRAM}$ (Section 2.1.2), the functions f_C defined using the transition relation $\rightsquigarrow$ in Figure 2.3:

$$\forall \xi \in \text{States}.\, f_C(\xi) = \bigsqcup \left\{ \{\!|\xi'|\!\} \mid C, \xi \rightsquigarrow \xi' \right\}$$

are local and satisfy (4.3). Note that the axioms in Figure 2.5 are sound with respect to the transformers defined in this way. Out of $f_C : \text{States} \to D$ for $C \in \mathsf{Seq}$ we can construct functions $g_C : \text{States}_\mathbf{c} \to D_\mathbf{c}$ defining the effect of primitive sequential commands on concrete states:

$$\forall \sigma \in \text{States}_\mathbf{c}.\, g_C(\sigma) = \beta(f_C(\beta^{-1}(\sigma))).$$

We lift the functions f_C and g_C to predicate transformers pointwise. We note for the future that (4.3) implies

$$\forall p \in D.\, \beta(f_C(p)) = g_C(\beta(p)). \tag{4.4}$$

Our semantics considers the commands from the set Seq atomic. Therefore, for each atomic command

$$C \in \mathsf{Seq} \cup \{\texttt{init}(E), \texttt{finalise}(E), \texttt{acquire}(E), \texttt{release}(E)\}$$

we consider the transition relation $\rightsquigarrow$ shown in Figure 4.7. Let $(N_k, T_k, \mathsf{start}_k, \mathsf{end}_k)$ be the CFG of thread k over the set of primitive commands

$$\mathsf{Seq} \cup \{\texttt{init}(E), \texttt{finalise}(E), \texttt{acquire}(E), \texttt{release}(E)\}$$

and let $N = \bigcup_{k=1}^n N_k$ and $T = \bigcup_{k=1}^n T_k$. As in the case of static locks, the interleaving operational semantics of the program S is given by a transition relation

$$\to_S : ((\{1, \dots, n\} \to N) \times \text{States}_\mathbf{c}) \times ((\{1, \dots, n\} \to N) \times (\text{States}_\mathbf{c} \cup \{\top\}))$$

defined by the rules in Figure 4.8. We adopt the same definitions of the initial and the final program counters, and the safety of programs as in Section 3.2.

In our proofs of soundness and formulation of an analysis, we use additional predicate transformers for lock-manipulating commands:

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad g_C(\{\sigma\}) \sqsubset \top \quad \sigma' \in g_C(\{\sigma\})}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \sigma'}$$

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad g_C(\{\sigma\}) = \top}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \top}$$

$$\frac{(v, C, v') \in T \quad C \notin \mathsf{Seq} \quad C, \sigma \rightsquigarrow \sigma'}{\mathsf{pc}[k : v], \sigma \rightarrow_S \mathsf{pc}[k : v'], \sigma'}$$

Figure 4.8: Operational semantics of concurrent programs with storable locks. The transition relation $\rightsquigarrow$ is defined in Figure 4.7.

$$\begin{array}{lcl}
\mathtt{init}_{A,\vec{F}}(E), (s, h[[\![E]\!]_s : \mathbf{Cell}(u)], \mathbf{i}) & \rightsquigarrow & (s, h[[\![E]\!]_s : \mathbf{Lock}(A, [\![\vec{F}]\!]_s, \mathsf{L}, 1)], \mathbf{i}) \\
\mathtt{finalise}(E), (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 1)], \mathbf{i}) & \rightsquigarrow & (s, h[[\![E]\!]_s : \mathbf{Cell}(u)], \mathbf{i}) \\
\mathtt{acquire}(E), (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{F}, \pi)], \mathbf{i}) & \rightsquigarrow & (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \\
\mathtt{acquire}(E), (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) & \not\rightsquigarrow & \\
\mathtt{release}(E), (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) & \rightsquigarrow & (s, h[[\![E]\!]_s : \mathbf{Lock}(A, \vec{w}, \mathsf{F}, \pi)], \mathbf{i}) \\
C, (s, h, \mathbf{i}) & \rightsquigarrow & \top, \text{ otherwise}
\end{array}$$

Figure 4.9: Transition relation on States for lock-manipulating commands. $\top$ indicates that the command faults. $\not\rightsquigarrow$ is used to denote that the command does not fault, but gets stuck.

- We define functions $g_C : \text{States}_{\mathbf{c}} \rightarrow D$ for

$$C \in \{\mathtt{init}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\}$$

  using the transition relation $\rightsquigarrow$ in Figure 4.7:

$$\forall \sigma \in \text{States}_{\mathbf{c}}.\, g_C(\sigma) = \bigsqcup \big\{ \{\!|\sigma'|\!\} \mid C, \sigma \rightsquigarrow \sigma' \big\}$$

  and let $g_{\mathtt{init}_{A,\vec{F}}(E)} = g_{\mathtt{init}(E)}$.

- We define functions $f_C : \text{States} \rightarrow D$ for

$$C \in \{\mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\}$$

  using the transition relation $\rightsquigarrow$ in Figure 4.9:

$$\forall \xi \in \text{States}.\, f_C(\xi) = \bigsqcup \big\{ \{\!|\xi'|\!\} \mid C, \xi \rightsquigarrow \xi' \big\}.$$

  Note that the functions f_C defined in this way are local.

We lift all the functions defined above to predicate transformers. Note that (4.4) holds for $C \in \{\mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\}$.

$\{\mathsf{x}, \mathsf{y} \Vdash \mathsf{x}{\mapsto}\_ * \mathsf{y}{\mapsto}\_\}$
$\mathtt{init}_{A,\mathsf{y}}\mathtt{(x);}$
$\mathtt{init}_{B,\mathsf{x}}\mathtt{(y);}$
$\{\mathsf{x}, \mathsf{y} \Vdash \mathsf{Lock}_A(\mathsf{x}, \mathsf{y}) * \mathsf{Hold}_A(\mathsf{x}, \mathsf{y}) * \mathsf{Lock}_B(\mathsf{y}, \mathsf{x}) * \mathsf{Hold}_B(\mathsf{y}, \mathsf{x})\}$
`release(x);`
$\{\mathsf{x}, \mathsf{y} \Vdash \mathsf{Lock}_A(\mathsf{x}, \mathsf{y}) * \mathsf{Hold}_B(\mathsf{y}, \mathsf{x})\}$
`release(y);`
$\{\mathsf{x}, \mathsf{y} \Vdash \mathsf{emp}_{\mathbf{h}}\}$

$$I_A(X, Y) = \mathsf{emp}_{\mathbf{s}} \land \mathsf{Lock}_B(Y, X) \quad \text{and} \quad I_B(X, Y) = \mathsf{emp}_{\mathbf{s}} \land \mathsf{Lock}_A(Y, X)$$

Figure 4.10: A pathological situation

4.5 Soundness of the logic

As it stands now, the logic allows some unpleasant situations to happen: in certain cases the proof system may not be able to detect a memory leak. Figure 4.10 shows an example of this kind. We assume defined lock sorts A and B with invariants $I_A(X, Y)$ and $I_B(X, Y)$. In this case the knowledge that the locks at the addresses `x` and `y` exist is lost by the proof system: the invariant for the lock `x` holds the full permission for the handle of the lock `y` and vice versa, hence, local states of the threads are then left without any permissions for the locks whatsoever.

Situations such as the one described above make the formulation of the soundness statement for our logic non-trivial. We first formulate a soundness statement (Theorem 4.1) showing that every final state of a program can be obtained as the $*$-conjunction of the postconditions of threads and the resource invariants for the free locks allocated in the state, where we existentially quantify the set of free locks, their sorts and parameters. Note that here a statement about a state uses the information about the free locks allocated in the same state. We then put restrictions on resource invariants that rule out situations similar to the one shown in Figure 4.10 and formulate another soundness statement (Theorem 4.4) in which the set of free locks in a final state is computed solely from the postconditions of threads.

We remind the reader that we distinguish between states States, local states $\text{States}_{\mathbf{l}}$ (Section 4.3), and concrete states $\text{States}_{\mathbf{c}}$ (Section 4.4), and the corresponding domains D, $D_{\mathbf{l}}$, and $D_{\mathbf{c}}$. Let $\text{LockParams} = \bigcup_{k \geq 0} (\mathcal{A}_{k+1} \times \text{Locs} \times \text{Values}^k)$ be the set of tuples of lock sorts and values of the corresponding lock parameters. We say that a state from $\text{States}_{\mathbf{c}}$ is *complete* if permissions associated with all the locks allocated in it are equal to 1 and their values to F or L. Note that according to the semantics in Section 4.4, if σ_0 is complete and $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, then σ is also complete. The first soundness theorem is formulated as follows.

Theorem 4.1 (Soundness of the logic: variant I). *Suppose $I \vdash \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$, where either*

- *the resource invariants in I are precise, the $*$ operation is cancellative, and* Init′ *is used instead of* Init *in the derivation of the triples; or*
- Conj *and* Forall *are not used in the derivation of the triples.*

Then for any complete state $\sigma_0 \in \mathrm{States}_{\mathbf{c}}$ such that for some $W_0 \subseteq \mathrm{LockParams}$

$$\sigma_0 \in \beta\left(\left(\mathop{\circledast}_{k=1}^{n} [\![P_k]\!]\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W_0} [\![I]\!]^{\mathsf{F}}_A(u,\vec{w})\right)\right),$$

the program S is safe when run from σ_0, and whenever $\mathsf{pc}_0, \sigma_0 \to^*_S \mathsf{pc}_{\mathbf{f}}, \sigma$, for some $W \subseteq \mathrm{LockParams}$ we have

$$\sigma \in \beta\left(\left(\mathop{\circledast}_{k=1}^{n} [\![Q_k]\!]\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W} [\![I]\!]^{\mathsf{F}}_A(u,\vec{w})\right)\right).$$

Note that by the definition of $[\![I]\!]^{\mathsf{F}}_A(u,\vec{w})$, the locks from W in the conclusion of the theorem are free in the state σ. We give the proof of the theorem in Section 4.5.1 below. We note that an analogue of the data-race freedom theorem for static locks (Corollary 3.16) also holds for storable locks.

We now proceed to formulate a soundness statement in which the component $\circledast_{(A,u,\vec{w})\in W} [\![I]\!]^{\mathsf{F}}_A(u,\vec{w})$ from Theorem 4.1 representing the resource invariants for free locks in the final state is obtained directly from the thread postconditions Q_k. To this end, we introduce an auxiliary notion of closure. Intuitively, closing a state from States amounts to $*$-conjoining it to the invariants of all free locks whose handles are reachable via resource invariants from the handles present in the state. For a state $\xi \in \mathrm{States}$ let $\mathsf{Free}(\xi)$, respectively, $\mathsf{Unknown}(\xi)$ be the subset of LockParams consisting of sorts and parameters of locks allocated in the state that have value F, respectively, U.

Definition 4.2 (Closure). *For $\mathcal{I} \in \mathrm{InvMaps}$ and $p \in \mathcal{P}(\mathrm{States})$ let $r \in \mathcal{P}(\mathrm{States})$ be the least predicate such that*

$$p \cup \left\{\xi_1 * \xi_2 \mid \xi_1 \in r \wedge \xi_2 \in \mathop{\circledast}_{(A,u,\vec{w})\in \mathsf{Unknown}(\xi_1)} \mathcal{I}^{\mathsf{F}}_A(u,\vec{w})\right\} \subseteq r.$$

The closure $\langle p \rangle_{\mathcal{I}}$ of p is the set of states from r that do not contain locks with the value U.

In general, the closure is not guaranteed to add invariants for all the free locks allocated in the state. For example, the closure of the postcondition of the program in Figure 4.10 still has an empty heap while in the final states obtained by executing the operational semantics there are locks allocated at addresses `x` and `y`. The problem is that there may exist a “self-contained” set of free locks (containing the locks at the addresses `x` and `y` in our example) such that the corresponding resource invariants hold full permissions for all the locks from the set. Local states of threads are then left without any permissions for the locks in the set, and hence, closure is not able to reach to their invariants. The following condition on resource invariants ensures that this does not happen.

Definition 4.3 (Admissibility of resource invariants). *Resource invariants in* $\mathcal{I} \in \text{InvMaps}$ *are admissible if there do not exist non-empty set* $W \subseteq \text{LockParams}$ *and state* $\xi \in \circledast_{(A,u,\vec{w})\in W} \mathcal{I}^{\mathsf{L}}_A(u,\vec{w})$ *such that for all* $(A,u,\vec{w}) \in W$ *the permission associated with the lock at the address* u *in* ξ *is* 1*.*

A resource invariant mapping I is admissible when its denotation $[\![I]\!]$ is. Revisiting Example 3 of Section 4.2, we can check that any state satisfying the closure of $[\![O \Vdash (1/M)H(\mathtt{head})]\!]$ represents an acyclic sorted list starting at `head`. It is easy to check that resource invariants for the set of lock sorts $\{R, P, H, N\}$ from Section 4.2 are admissible whereas those for $\{A, B\}$ from this section are not. The admissibility of N is due to the fact that I_N implies sortedness of lists built out of resource invariants for N, hence, the invariants cannot form a cycle.

We can now formulate the second soundness statement that lets us check the absence of memory leaks.

Theorem 4.4 (Soundness of the logic: variant II). *Suppose* $I \vdash \{P_k\}\ C_k\ \{Q_k\}$ *for* $k = 1..n$ *and the restrictions on the derivations from Theorem 4.1 hold. Suppose further that either at least one of* Q_k *is intuitionistic or the resource invariants in* I *are admissible. Then for any complete state* $\sigma_0 \in \text{States}_{\mathsf{c}}$ *such that*

$$\sigma_0 \in \beta\left(\left\langle \circledast_{k=1}^{n} [\![P_k]\!] \right\rangle_{[\![I]\!]}\right),$$

the program S *is safe when run from* σ_0*, and whenever* $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}_{\mathsf{f}}, \sigma$*, we have*

$$\sigma \in \beta\left(\left\langle \circledast_{k=1}^{n} [\![Q_k]\!] \right\rangle_{[\![I]\!]}\right).$$

Proof. Let $\mathcal{I} = [\![I]\!]$. Consider a complete state $\sigma_0 \in \beta(\langle \circledast_{k=1}^n [\![P_k]\!]\rangle_{\mathcal{I}})$. From the definition of closure it follows that for some $W_0 \subseteq \text{LockParams}$ we have $\sigma_0 \in \beta\left((\circledast_{k=1}^n [\![P_k]\!]) * \left(\circledast_{(A,u,\vec{w})\in W_0} \mathcal{I}^{\mathsf{F}}_A(u,\vec{w})\right)\right)$. Then by Theorem 4.1 the program S is safe when run from σ_0 and if $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}_{\mathsf{f}}, \sigma$, then for some $\xi \in \text{States}$ we have $\sigma = \beta(\xi)$ and $\xi \in (\circledast_{k=1}^n [\![Q_k]\!]) * \left(\circledast_{(A,u,\vec{w})\in \mathsf{Free}(\xi)} \mathcal{I}^{\mathsf{F}}_A(u,\vec{w})\right)$. Hence, by the definition of closure, we have $\xi \in \xi_1 * \xi_2$ where $\xi_1 \in \langle \circledast_{k=1}^n [\![Q_k]\!]\rangle_{\mathcal{I}}$ and $\xi_2 \in \circledast_{(A,u,\vec{w})\in W} \mathcal{I}^{\mathsf{F}}_A(u,\vec{w})$ for some $W \subseteq \mathsf{Free}(\xi)$. If one of Q_k is intuitionistic, then from this it directly follows that $\sigma \in \beta(\langle \circledast_{k=1}^n [\![Q_k]\!]\rangle_{\mathcal{I}})$.

Suppose now that $W \neq \emptyset$ and the resource invariants for lock sorts mentioned in W are admissible. Consider any $(A,u,\vec{w}) \in W$. The state σ is complete, therefore, the permission associated with the lock at the address u in ξ is 1. Besides, since $W \subseteq \mathsf{Free}(\xi)$, the value associated with u in ξ is F. Hence, if the permission associated with u in ξ_2 were less than 1, then u would have to be allocated in ξ_1 with a non-zero permission and the value U, which would contradict the definition of closure (a state in a closure cannot contain locks with the value U). So, for any $(A, u, \vec{w}) \in W$ the permission associated with u in ξ_1 is 1, which contradicts the admissibility of resource invariants for lock sorts used in the proof of the program. Therefore, $W = \emptyset$, which implies $\sigma \in \beta\left(\langle \circledast_{k=1}^{n} [\![Q_k]\!] \rangle_{\mathcal{I}}\right)$. $\square$

Note that for garbage-collected languages we can use the intuitionistic version of the logic [45] (i.e., one in which every assertion is intuitionistic) and, hence, do not have to check admissibility. Also, admissibility does not have to be checked if we are not interested in detecting memory leaks, as then Theorem 4.1 can be used.

4.5.1 Proof of soundness

The proof is done according to the method of Section 3.4. As in that case, we prove the soundness of the logic with the aid of a thread-local interpretation formulated using semantic proofs—triples $(C, G, \mathcal{I})$, where

- C is a command with a CFG $(N, T, \mathsf{start}, \mathsf{end})$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{init}(E), \mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\}$;
- $G : N \to D_{\mathrm{l}}$ maps program points of C to semantic annotations;
- $\mathcal{I} \in \mathrm{InvMaps}$ is a semantic resource invariant mapping

such that for all edges $(v, C', v') \in T$

- if $C' \in \mathsf{Seq} \cup \{\mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E)\}$, then
$$f_{C'}(G(v)) \sqsubseteq G(v'), \tag{4.5}$$
where $f_{C'}$ for $C' \in \{\mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E)\}$ are defined in Section 4.4;
- if C' is $\mathtt{acquire}(E)$, then
$$\begin{gathered} G(v') \neq \top \Rightarrow G(v) \neq \top \wedge \forall \xi \in G(v).\, \exists s, h, \mathbf{i}, u, A, \vec{w}, b, \pi. \\ \xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, b, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \{\xi\} * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}) \sqsubseteq G(v'); \end{gathered} \tag{4.6}$$
- if C' is $\mathtt{release}(E)$, then
$$\begin{gathered} G(v') \neq \top \Rightarrow G(v) \neq \top \wedge \forall \xi \in G(v).\, \exists s, h, \mathbf{i}, u, A, \vec{w}, \pi. \\ \xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \xi \in G(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}); \end{gathered} \tag{4.7}$$

- if C' is $\texttt{init}(E)$, then

$$G(v') \neq \top \Rightarrow G(v) \neq \top \wedge \forall \xi \in G(v).\, \exists s, h, \mathbf{i}, u, b, A, \vec{w}.$$
$$\xi = (s, h[u : \mathbf{Cell}(b)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 1)], \mathbf{i}) \in G(v'). \quad (4.8)$$

Note that we treat annotated and unannotated versions of the `init` command differently, since in the former case we need to record the fact that the command behaves consistently with the annotation in the semantic proof. The definition of validity $I \vDash \{P\}\ C\ \{Q\}$ with respect to the thread-local interpretation repeats Definition 3.5.

As before, inequalities (4.5)–(4.8) mimic the corresponding axioms, in particular, the axioms are sound with respect to the thread-local interpretation. Additionally, the inequalities are crafted in such a way that an analogue of Lemma 3.8 holds for the semantic proofs defined here (Lemma 4.7 below), which allows us to justify the soundness of the rules Frame, Disj, and Conj.

The following analogue of Lemma 3.1 establishes a correspondence between the thread-local interpretation and the global semantics of Section 4.4.

Lemma 4.5 (Parallel Decomposition Lemma). *Assume semantic proofs* $(C_k, G_k, \mathcal{I})$, $k = 1..n$. *If a complete state* $\sigma_0 \in \text{States}_{\mathbf{c}}$ *is such that for some* $W_0 \subseteq \text{LockParams}$

$$\{\sigma_0\} \sqsubseteq \beta\left(\left(\mathop{\circledast}_{k=1}^{n} G_k(\mathsf{start}_k)\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W_0} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right), \quad (4.9)$$

then, whenever $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$, *for some* $W \subseteq \text{LockParams}$ *we have*

$$\{\!|\sigma|\!\} \sqsubseteq \beta\left(\left(\mathop{\circledast}_{k=1}^{n} G_k(\mathsf{pc}(k))\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right). \quad (4.10)$$

Proof. We prove the statement of the theorem by induction on the length of the derivation of σ. In the base case (4.10) is equivalent to (4.9). Suppose now that

$$\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}[j : v], \sigma \rightarrow_S \mathsf{pc}[j : v'], \sigma'.$$

Then $(v, C, v') \in T$ for some atomic command C. We have to show that if for some $W \subseteq \text{LockParams}$

$$\{\sigma\} \sqsubseteq \beta\left(\left(\mathop{\circledast}_{k=1}^{n} G_k((\mathsf{pc}[j : v])(k))\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right), \quad (4.11)$$

then for some $W' \subseteq \text{LockParams}$

$$\{\!|\sigma'|\!\} \sqsubseteq \beta\left(\left(\mathop{\circledast}_{k=1}^{n} G_k((\mathsf{pc}[j : v'])(k))\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W'} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right). \quad (4.12)$$

There are three cases corresponding to the type of the command C.

Case 1. $C \in \mathsf{Seq} \cup \{\mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E)\}$. Let

$$r = \left(\mathop{\circledast}_{\substack{1 \le k \le n, \\ k \ne j}} G_k(\mathsf{pc}(k)) \right) * \left(\mathop{\circledast}_{(A,u,\vec{w}) \in V} \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) \right), \tag{4.13}$$

where $V = W$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &\sqsubseteq g_C(\{\sigma\}) && \text{definition of } \rightarrow_S \\ &\sqsubseteq g_C(\beta(G_j(v) * r)) && (4.11) \\ &= \beta(f_C(G_j(v) * r)) && (4.4) \\ &\sqsubseteq \beta(f_C(G_j(v)) * r) && f_C \text{ is local} \\ &\sqsubseteq \beta(G_j(v') * r) && (4.5) \end{aligned}$$

which is equivalent to (4.12) with $W' = W$.

Case 2. C *is* $\mathtt{acquire}(E)$. Given (4.11), we can assume that

$$\{\sigma\} \sqsubseteq \beta(\{\xi\} * \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) * r), \tag{4.14}$$

where

$$\{\xi\} \sqsubseteq G_j(v), \tag{4.15}$$

$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{U}, \pi)], \mathbf{i})$, $[\![E]\!]_s = u$, $(A, u, \vec{w}) \in W$, and r is defined by (4.13) with $V = W \backslash \{(A, u, \vec{w})\}$; otherwise the right-hand side of (4.12) is $\top$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &\sqsubseteq g_C(\{\sigma\}) && \text{definition of } \rightarrow_S \\ &\sqsubseteq g_C(\beta(\{\xi\} * \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) * r)) && (4.14) \\ &= \beta(f_C(\{\xi\} * \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) * r)) && (4.4) \\ &= \beta(\{\xi\} * \mathcal{I}^{\mathsf{L}}_A(u, \vec{w}) * r) && \text{definition of } f_C \\ &\sqsubseteq \beta(G_j(v') * r) && (4.6) \text{ and } (4.15) \end{aligned}$$

which is equivalent to (4.12) with $W' = W \backslash \{(A, u, \vec{w})\}$.

Case 3. C *is* $\mathtt{release}(E)$. We can assume that

$$\{\sigma\} \sqsubseteq \beta(\{\xi\} * r), \tag{4.16}$$

where (4.15) holds, $\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i})$, $[\![E]\!]_s = u$, $(A, u, \vec{w}) \notin W$, and r is defined by (4.13) with $V = W$; otherwise the right-hand side of (4.12) is $\top$. Then

$$\begin{aligned} \{\!|\sigma'|\!\} &\sqsubseteq g_C(\{\sigma\}) && \text{definition of } \rightarrow_S \\ &\sqsubseteq g_C(\beta(\{\xi\} * r)) && (4.16) \\ &= \beta(f_C(\{\xi\} * r)) && (4.4) \\ &\sqsubseteq \beta(f_C(G_j(v') * \mathcal{I}^{\mathsf{L}}_A(u, \vec{w}) * r)) && (4.7) \text{ and } (4.15) \\ &= \beta(G_j(v') * \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) * r) && \text{definition of } f_C \end{aligned}$$

which is equivalent to (4.12) with $W' = W \cup \{(A, u, \vec{w})\}$.

Case 4. C is `init`(E). We can assume that

$$\{\sigma\} \sqsubseteq \beta(\{(s, h[u : \mathbf{Cell}(b)], \mathbf{i})\} * r), \tag{4.17}$$

where $(s, h[u : \mathbf{Cell}(b)], \mathbf{i}) \in G(v)$, $[\![E]\!]_s = u$, and r is defined by (4.13) with $V = W$; otherwise the right-hand side of (4.12) is $\top$. Then by (4.8)

$$\{(s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 1)], \mathbf{i})\} \sqsubseteq G(v') \tag{4.18}$$

for some A and $\vec{w}$. We have:

$$\begin{array}{lll} \{\!|\sigma'|\!\} & \sqsubseteq g_C(\{\sigma\}) & \text{definition of } \rightarrow_S \\ & \sqsubseteq g_C(\beta(\{(s, h[u : \mathbf{Cell}(b)], \mathbf{i})\} * r)) & (4.17) \\ & = \beta(\{(s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 1)], \mathbf{i})\} * r) & \text{definition of } g_C \\ & \sqsubseteq \beta(G(v') * r) & (4.18) \end{array}$$

which is equivalent to (4.12) with $W' = W$. □

Lemma 4.6. *The axioms* Prim, Init, Init′, Finalise, Acquire, *and* Release *are sound with respect to the thread-local interpretation.*

To justify the soundness of the rules Frame, Disj, and Conj, we have to establish an analogue of Lemma 3.8.

Lemma 4.7. *(i) For any $r \in D$, if $(C, G, \mathcal{I})$ is a semantic proof, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G(v) * r$.*

(ii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \sqcup G_2(v)$.

(iii) If $(C, G_1, \mathcal{I})$ and $(C, G_2, \mathcal{I})$ are semantic proofs, then so is $(C, G', \mathcal{I})$, where $\forall v.\, G'(v) = G_1(v) \sqcap G_2(v)$, provided the resource invariants in $\mathcal{I}$ are precise, the $*$ operation is cancellative, and all `init` *commands in C are annotated with lock sorts and parameters.*

Proof. Take an edge (v, C', v') in the CFG of the command C. We only consider the case when C' is `release`(E) (the others are treated analogously).

(i) Suppose (4.7) holds. We have to show that (4.7) also holds for $G = G'$. To this end, assume $G'(v') \neq \top$, then $G(v') \neq \top$ and $r \neq \top$. By (4.7), we have $G(v) \neq \top$, thus, $G'(v) \neq \top$. Consider $\xi \in G(v) * r$. We have $\xi = \xi_1 * \xi_2$, where $\xi_1 \in G(v)$ and $\xi_2 \in r$. Since $G(v') \neq \top$, from (4.7) we get

$$\xi_1 = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \xi_1 \in G(v') * \mathcal{I}^{\mathsf{L}}_A(u, \vec{w}).$$

Then

$$\xi = (s', h'[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_{s'} = u \wedge \xi \in G(v') * r * \mathcal{I}^{\mathsf{L}}_A(u, \vec{w})$$

as required.

(ii) Suppose (4.7) holds for $G = G_1$ and $G = G_2$. We have to show that (4.7) also holds for $G = G'$. Assume $G'(v') \neq \top$, then $G_1(v') \neq \top$ and $G_2(v') \neq \top$. By (4.7), we have $G_1(v) \neq \top$ and $G_2(v) \neq \top$, thus, $G'(v) \neq \top$. Consider $\xi \in G_1(v) \sqcup G_2(v)$, then either $\xi \in G_1(v)$ or $\xi \in G_2(v)$. Since $G_1(v') \neq \top$ and $G_2(v') \neq \top$, from (4.7) for $G = G_1$ and $G = G_2$ we get

$$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \\ (\xi \in G_1(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}) \vee \xi \in G_2(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w})),$$

which entails $\xi \in (G_1(v') \sqcup G_2(v')) * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w})$ as required.

(iii) Suppose (4.7) holds for $G = G_1$ and $G = G_2$. Assume $G'(v') \neq \top$, then either $G_1(v') \neq \top$ or $G_2(v') \neq \top$. By (4.7), we have $G_1(v) \neq \top$ or $G_2(v) \neq \top$, thus, $G'(v) \neq \top$.

Suppose first that $G_1(v') \neq \top$ and $G_2(v') \neq \top$ and consider $\xi \in G_1(v) \sqcap G_2(v)$, then $\xi \in G_1(v)$ and $\xi \in G_2(v)$. From (4.7) for $G = G_1$ and $G = G_2$ we get

$$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \\ \xi \in G_1(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}) \wedge \xi \in G_2(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}).$$

Since $\mathcal{I}_A^{\mathsf{L}}(u, \vec{w})$ is precise and the $*$ operation is cancellative, this entails $\xi \in (G_1(v') \sqcap G_2(v')) * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w})$ as required.

Assume now that $G_1(v') \neq \top$ and $G_2(v') = \top$. From (4.7) for $G = G_1$ we get

$$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \xi \in G_1(v') * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w}).$$

Since $G_1(v') = G_1(v') \sqcap \top$, this entails $\xi \in (G_1(v') \sqcap G_2(v')) * \mathcal{I}_A^{\mathsf{L}}(u, \vec{w})$ as required. The case when $G_1(v') = \top$ and $G_2(v') \neq \top$ is treated analogously. $\square$

We note that an analogue of Lemma 3.11 justifying the soundness of the rules Exists and Forall holds for semantic proofs introduced here (in the case of Forall only under the assumptions of Lemma 4.7(iii)). We can now formulate an analogue of Lemma 3.10, ensuring soundness of the logic with respect to the thread-local interpretation.

Lemma 4.8. *If $I \vdash \{P\}\ C\ \{Q\}$ and the restrictions on the derivation from Theorem 4.1 hold, then $I \models \{P\}\ C\ \{Q\}$.*

The proof is done by induction on the derivation of the triple using Lemmas 4.6 and 4.7, and analogues of Lemmas 3.7 and 3.11. The proof of Theorem 4.1 then literally repeats the proof of Theorem 3.4 with Lemma 4.8 used instead of Lemma 3.10. As in the case of static locks, Theorem 3.4 also holds when derivations use axioms in Figure 2.5 for commands from SeqRAM.

4.6 Deriving the analysis

From the logic presented in this chapter and its thread-local interpretation defined in Section 4.5.1, we can derive a scheme of a thread-modular analysis for concurrent programs with storable locks similar to the one of Chapter 3.

Analysis formulation. We assume an abstract interpretation with state separation (Section 3.3) with an abstract separation domain $D_{\mathrm{l}}^\sharp$, the local concrete domain D_{l} of Section 4.3, a concretisation function $\gamma : D_{\mathrm{l}}^\sharp \to D_{\mathrm{l}}$, and abstract transfer functions $f_C^\sharp : D_{\mathrm{l}}^\sharp \to D_{\mathrm{l}}^\sharp$ for $C \in \mathsf{Seq}$ over-approximating the corresponding predicate transformers $f_C : D \to D$ of Section 4.4:

$$\forall p \in D_{\mathrm{l}}^\sharp.\, f_C(\gamma(p)) \sqsubseteq \gamma(f_C^\sharp(p)). \tag{4.19}$$

We further assume a set of lock sorts $\mathcal{A}$ with fixed arities and parameters. We denote with InvMaps$^\sharp$ the set of abstract resource invariant mappings—functions $I^\sharp : \mathcal{A} \to D^\sharp$ such that for any $A \in \mathcal{A}$ we have $\gamma(I_A^\sharp) \in D_{\mathrm{l}}[\vec{Y}]$, where $\vec{Y}$ are the parameters of A. We lift the concretisation function γ to InvMaps$^\sharp$ as follows: for any $A \in \mathcal{A}$ we let $(\gamma(I^\sharp))_A = \gamma(I_A^\sharp)$. Thus, $\gamma(I^\sharp) \in$ InvMaps. We also assume that the join operator on $D_{\mathrm{l}}^\sharp$ is such that

$$\forall p, q \in D_{\mathrm{l}}^\sharp.\, \gamma(p) \in D_{\mathrm{l}}[\vec{Y}] \wedge \gamma(q) \in D_{\mathrm{l}}[\vec{Y}] \Rightarrow \gamma(p \sqcup q) \in D_{\mathrm{l}}[\vec{Y}]. \tag{4.20}$$

This ensures that computing the join of two approximations of the resource invariant for a lock sort yields an approximation of the invariant for the same lock sort.

The analysis operates on the domain $\widehat{D}^\sharp = (N \to D_{\mathrm{l}}^\sharp) \times \text{InvMaps}^\sharp$, where the first component gives the local states of the threads and the second the resource invariants for the corresponding lock sorts. It is defined by the functional $\mathcal{F}^\sharp : \widehat{D}^\sharp \to \widehat{D}^\sharp$ in Figure 4.11. The overall scheme of the analysis is similar to that of the analysis for static locks (Figure 3.3), except now the transfer functions $g_C^\sharp$ follow the treatment of commands in the thread-local interpretation of Section 4.5.1. The analysis is parameterised by the following functions defining processing of `init`, `acquire`, `release`, and `finalise`:

- $f_C^\sharp : D_{\mathrm{l}}^\sharp \to D_{\mathrm{l}}^\sharp$ for $C = \mathtt{init}(E)$, such that an analogue of (4.8) holds:

$$\forall p \in D_{\mathrm{l}}^\sharp.\, \gamma(f_C^\sharp(p)) \neq \top \Rightarrow p \neq \top \wedge \forall \xi \in p.\, \exists s, h, \mathbf{i}, u, b, A, \vec{w}.$$
$$\xi = (s, h[u : \mathbf{Cell}(b)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, 1)], \mathbf{i}) \in \gamma(f_C^\sharp(p)).$$

- $f_C^\sharp : D_{\mathrm{l}}^\sharp \to D_{\mathrm{l}}^\sharp$ for $C = \mathtt{finalise}(E)$, such that (4.19) holds for f_C defined in Section 4.4.

- $f_C^\sharp : \text{InvMaps}^\sharp \times D_{\mathrm{l}}^\sharp \to D_{\mathrm{l}}^\sharp$ for $C = \mathtt{acquire}(E)$ satisfying an analogue of (4.6):

$$\forall p \in D_{\mathrm{l}}^\sharp.\, \gamma(f_C^\sharp(p, I^\sharp)) \neq \top \Rightarrow p \neq \top \wedge \forall \xi \in p.\, \exists s, h, \mathbf{i}, u, A, \vec{w}, b, \pi.$$
$$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, b, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge \{\xi\} * (\gamma(I^\sharp))_A(u, \vec{w}) \sqsubseteq \gamma(f_C^\sharp(p, I^\sharp)).$$

$\mathcal{F}^\sharp(G^\sharp, I^\sharp) = (\widetilde{G}^\sharp, \widetilde{I}^\sharp)$, where

- $\widetilde{G}^\sharp(\mathsf{start}_k) = e^\sharp$, $k = 1..n$;
- $\widetilde{G}^\sharp(v') = \bigsqcup_{(v,C,v') \in T} g^\sharp_C(G^\sharp(v))$
  for every program point $v' \in N \setminus \{\mathsf{start}_k \mid k = 1..n\}$, where

$$g^\sharp_C(p) = \begin{cases} f^\sharp_C(p), & \text{if } C \in \mathsf{Seq} \cup \{\mathtt{init}(E), \mathtt{finalise}(E)\}; \\ f^\sharp_C(I^\sharp, p), & \text{if } C \text{ is } \mathtt{acquire}(E); \\ \mathsf{ThreadLocal}_E(p), & \text{if } C \text{ is } \mathtt{release}(E); \end{cases}$$

- $\widetilde{I}^\sharp_A = \bigsqcup_{(v,\mathtt{release}(E),v') \in T} \mathsf{Protected}_E(G^\sharp(v), A)$ for every lock sort $A \in \mathcal{A}$.

Figure 4.11: Thread-modular analysis for concurrent programs with storable locks

This function would typically be constructed using the $*^\sharp$ operation on the abstract separation domain $D^\sharp_\mathfrak{l}$.

- A family of functions $\mathsf{ThreadLocal}_E : D^\sharp_\mathfrak{l} \to D^\sharp_\mathfrak{l}$ and $\mathsf{Protected}_E : D^\sharp_\mathfrak{l} \to (\mathcal{A} \to D^\sharp_\mathfrak{l})$ for every expression E defining the processing of $\mathtt{release}(E)$. The part of the heap given by $\mathsf{Protected}_E(p, A)$ is added to the resource invariant for the lock sort A. The part given by $\mathsf{ThreadLocal}_E(p)$ becomes the new local state of the thread. We require that $\mathsf{ThreadLocal}_E$ and $\mathsf{Protected}_E$ split the state soundly, formalised using an analogue of (4.7):

$$\forall p \in D^\sharp_\mathfrak{l}.\, \gamma(\mathsf{ThreadLocal}_E(p)) \neq \top \Rightarrow p \neq \top \wedge \forall \xi \in p.\, \exists s, h, \mathbf{i}, u, A, \vec{w}, \pi.$$
$$\xi = (s, h[u : \mathbf{Lock}(A, \vec{w}, \mathsf{L}, \pi)], \mathbf{i}) \wedge [\![E]\!]_s = u \wedge$$
$$\xi \in \gamma(\mathsf{ThreadLocal}_E(p)) * (\gamma(\mathsf{Protected}_E(p)))_A(u, \vec{w}).$$

We also require that $\forall p \in D^\sharp_\mathfrak{l}.\, \gamma(\mathsf{Protected}_E(p, A)) \in D_\mathfrak{l}[\vec{Y}]$, where $\vec{Y}$ are the parameters of A, i.e., that $\mathsf{Protected}_E$ produces an approximation of the resource invariant for the corresponding lock sort.

As in Chapter 3, we assume, for simplicity, that the initial local state of every thread is the empty heap. Note also that we do not require initial approximations of resource invariants, since in our programming language data structure initialisation code is a part of the code of threads.

A heuristic for determining heap splittings. A typical pattern in coarse-grained programs is for locks to be stored in structures, where some fields in a structure point to the data structure protected by the lock and are themselves protected by it. Usually, locks in the structures of the same type protect the same kind of data structures and, hence, can be assigned the same resource invariant. Moreover, parameterising resource invariants only with the address of the lock is enough for verifying memory safety of a wide range of programs (the general case is useful for handling dynamic thread creation; see Section 6.3). We can thus guess lock sorts out of type annotations in the program being analysed. In the pattern described above we call the fields pointing to the protected data structure the *entry fields* for the corresponding lock sort. Entry fields can be inferred by tools such as LOCKSMITH [63]. We can then define $\mathsf{Protected}_E(p, A)$ as computing the part of p reachable from the entry fields in the structure containing the lock at the address E, assuming the lock has the sort A; otherwise, $\mathsf{Protected}_E(p, A)$ can return $\bot$. $\mathsf{ThreadLocal}_E(p)$ is then the rest of p. $\mathsf{Protected}_E(p, A)$ also has to check that the corresponding Hold-fact is present in p.

Note that for the thread-modular analysis to handle linked data structures storing locks (e.g., a singly-linked list, where each node contains a pointer to a cyclic doubly-linked list and a lock protecting it), the underlying sequential heap analysis has to be able to summarise such data structures regardless of the sorts of the locks stored in them. An adaptive heap analysis, such as CDS (Section 2.2.1), is suitable for this purpose.

Soundness. To simplify stating the soundness of the analysis, we assume that the concretisation $\gamma(e^\sharp)$ of the unit element of $D_{\mathbf{l}}^\sharp$ does not contain any allocated locks.

Theorem 4.9 (Soundness of the analysis: variant I). *Let $(G^\sharp, I^\sharp)$ be a fixed point of the functional $\mathcal{F}^\sharp$ (then from (4.20) it follows that $\gamma(I^\sharp) \in \mathrm{InvMaps}$). If a complete state $\sigma_0 \in \mathrm{States}_{\mathbf{c}}$ is such that*

$$\{\sigma_0\} \sqsubseteq \beta\left(\mathop{\circledast}_{k=1}^{n} \gamma(e^\sharp)\right),$$

then whenever $\mathsf{pc}_0, \sigma_0 \to_S^* \mathsf{pc}, \sigma$, for some $W \subseteq \mathrm{LockParams}$ we have

$$\{\!|\sigma|\!\} \sqsubseteq \beta\left(\left(\mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp(\mathsf{pc}(k)))\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W} (\gamma(I^\sharp))_A^{\mathsf{F}}(u, \vec{w})\right)\right).$$

The proof is similar to the proof of Theorem 3.2. As in that case, we can show that the result of the analysis generates an instance of the thread-local interpretation of Section 4.5.1 with the local states $\gamma(G^\sharp)$ and the semantic resource invariant mapping $\gamma(I^\sharp)$.

We note that analogues of Corollaries 3.3 and 3.15 can be proved using the above soundness theorem, thus, we can use the analysis to check memory safety and data-race freedom of programs. Theorem 4.9 does not allow us, however, to check the absence of memory leaks as explained in Section 4.5. This can be done using the following theorem, analogous to Theorem 4.4. Let us extend the definition of closure (Definition 4.2) to the domain D by letting $\langle\top\rangle = \top$.

Theorem 4.10 (Soundness of the analysis: variant II). *Let $(G^\sharp, I^\sharp)$ be a fixed point of the functional $\mathcal{F}^\sharp$ (then from (4.20) it follows that $\gamma(I^\sharp) \in \mathrm{InvMaps}$) and $\sigma_0 \in \mathrm{States}_\mathbf{c}$ a complete state such that*

$$\{\sigma_0\} \sqsubseteq \beta\left(\mathop{\circledast}_{k=1}^{n} \gamma(e^\sharp)\right).$$

If $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$ and either $\gamma(G^\sharp(\mathsf{pc}(k)))$ is intuitionistic for some k or the resource invariants in $\gamma(I^\sharp)$ are admissible, then

$$\{\!\!\{\sigma\}\!\!\} \sqsubseteq \beta\left(\left\langle \mathop{\circledast}_{k=1}^{n} \gamma(G^\sharp(\mathsf{pc}(k)))\right\rangle_{\gamma(I^\sharp)}\right).$$

The proof is the same as for Theorem 4.4.

Admissibility is a semantic condition and is hard to check automatically. For coarse-grained programs, we can check the following condition implying admissibility. Given a resource invariant mapping, let us define a directed graph with lock sorts as vertices, where there is an edge from A to B if the resource invariant for the lock sort A may contain a handle for a lock of the sort B. If the graph is acyclic, then the resource invariants are admissible. The existence of an edge between two lock sorts in the graph can be easily checked in abstract domains for heap analysis based on separation logic (Section 2.2.1).

4.7 Related work

An advantage of the logic for storable locks presented in this chapter is its simple semantics, which, additionally, yields a scheme of the corresponding program analysis. A disadvantage of the logic is that we cannot use it to give specifications to code that is generic in resource invariants, since we always have to name the sort of every lock an assertion talks about. This problem has been resolved by Hobor et al. [43], who suggested a logic for storable locks with a more powerful assertion language. Their semantics uses step-indexing to resolve the paradox arising in (4.1), however, this solution drastically complicates the semantics of the logic and its proof of soundness.

Leino and Müller [47] have recently proposed a logic for storable locks that is virtually identical to the one presented here, but encoded into the first-order logic of the Boogie program verifier [1]. They use the logic as a basis for an assertion-based verifier of concurrent programs.

Locks that our logic reasons about are not re-entrant: a thread that tries to acquire the same lock twice deadlocks. Haack et al. [38] have extended the logic to handle re-entrant locks.

Feng et al. [27] and Vafeiadis and Parkinson [77, 75] have recently suggested combinations of separation logic and rely-guarantee reasoning that, among other things, can be used to reason about storable locks. One of them [77] served as a basis for an assertion checker [16] and a heap analysis [76] (the latter discussed in Section 3.6). In [77] locks are not treated natively in the logic, but are represented as cells in memory holding the identifier of the thread that holds the lock; rely-guarantee is then used to simplify reasoning about the global shared heap with locks allocated in it. The logic allows modular reasoning about complex fine-grained concurrent algorithms (e.g., about the optimistic list mentioned in Section 4.2), but loses the locality of reasoning for programs that allocate and deallocate many simple data structures protected by locks, which results in awkward proofs. In other words, as the original concurrent separation logic, the logics in [27, 77, 75] are designed for reasoning about the concurrent control of bounded numbers of data structures whereas our logic is designed to reason about the concurrent control of unboundedly many data structures that are dynamically created and destroyed. Ideally, one wants to have a combination of both: a logic in which on the higher-level the reasoning is performed in a resource-oriented fashion and on the lower-level rely-guarantee is applied to deal with complex cases. Designing such a logic is ongoing work [26, 25, 23].

Chapter 5

Procedures

In this chapter we consider interprocedural heap analyses, which target programs with (possibly recursive) first-order procedures. Interprocedural analyses for sequential programs usually compute procedure summaries that approximate the semantics of a procedure by associating representations of the program state at procedure entry to corresponding result states at procedure exit. In program logics, the analogue of procedure summaries are procedure specifications. For example, a proof of a program with procedures in separation logic may use the procedure specification

$$\{\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}\ \mathtt{append}\ \{\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}$$

for a procedure that destructively appends the list with the head pointed to by the global variable `x` to the list with the head pointed to by the global variable `y` (we assume a programming language with parameterless procedures; procedures with parameters are represented as syntactic sugar by passing parameters via global variables). The benefit of summaries for interprocedural program analysis is that, once a summary is computed, it can be stored and subsequently reused if the procedure is called again in the same calling context, thus making the analysis more efficient. The main obstacle to scaling interprocedural analyses to larger and more complicated programs is the explosion in the number of calling contexts the analysis has to consider. This problem is especially acute for heap analyses because of large sizes of the corresponding abstract domains. We can avoid the problem by performing context-insensitive analysis that computes a single summary for each procedure, joining the result states for all calling contexts. Unfortunately, for heap analysis this leads to overly imprecise results—in this case we need the analysis to be context-sensitive.

One way to reduce the number of calling contexts in heap analyses is to perform localisation at procedure calls, i.e., to pass only the relevant part of the program state to the procedure [67]. For example, the procedure `append` may be called in two different contexts, described by the formulae

$$\mathtt{x}, \mathtt{y}, \mathtt{u} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL})$$

and

$$\mathtt{x}, \mathtt{y}, \mathtt{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathtt{v} \mapsto \mathtt{NULL},$$

where `u` and `v` are local variables of the callers. When the procedure is called in the first context, we can:

- split the heap into a *local heap* describing the part relevant to the procedure (e.g., $\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$) and a *frame* (e.g., $\mathtt{u} \Vdash \mathsf{ls}(\mathtt{u}, \mathtt{NULL})$);
- analyse the procedure on the local heap obtaining $\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$ as the post-heap;
- conjoin the post-heap with the frame yielding the resulting heap $\mathtt{x}, \mathtt{y}, \mathtt{u} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL})$.

From the perspective of separation logic, this corresponds to applying the frame rule over the procedure call:

$$\frac{\{\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}\ \mathtt{append}\ \{\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}}{\begin{array}{c}\{\mathtt{x}, \mathtt{y}, \mathtt{u} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL})\} \\ \mathtt{append} \\ \{\mathtt{x}, \mathtt{y}, \mathtt{u} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL})\}\end{array}}$$

When the same algorithm is performed in the second context with the local heap $\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$ and the frame $\mathtt{v} \Vdash \mathtt{v} \mapsto \mathtt{NULL}$, we are able to reuse the results of analysing the procedure in the first context.

Note that the choice of the splitting of the heap into a local heap and a frame is not important for soundness: any splitting is sound, but if too small a local heap is chosen, the analysis will discover a false memory safety error while analysing the procedure. One possible way to split the heap is to take as the local heap the part reachable from the actual parameters of the procedure and the global variables [67], which in our desugaring of parameter passing corresponds to just global variables (this is how the heap splittings in the above examples are obtained). Note that this heuristic may sometimes carve out too big a local heap, making the analysis less efficient: e.g., passing only $\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL})$ is sufficient for most implementations of `append`. However, it works well in practice.

In this chapter, we develop a framework for constructing context-sensitive interprocedural analyses with localisation for sequential programs out of intraprocedural analyses operating on arbitrary abstract separation domains (Section 5.3) that generalises the Reps-Horwitz-Sagiv (RHS) algorithm for interprocedural analysis [64]. This extends the existing work on interprocedural analysis of sequential heap-manipulating programs using RHS-based analyses [67, 70, 31, 81], which so far has considered only particular abstract domains or localisation schemes. We also present an instantiation of our framework with an abstract domain based on separation logic, which at the time of its publication [31] outperformed earlier interprocedural heap analyses [70, 69] and has since been used as a component in other analyses [81].

In Section 5.2, we propose an abstract version of separation logic for programs with procedures, where local procedure environments and global heaps are represented by arbitrary separation algebras. Our interprocedural analysis can be viewed as generating proofs in the logic, so that the localisation step corresponds to the frame rule as described above. We give two proofs of soundness to the logic. The first one is an elegant proof using standard techniques that establishes the soundness of the whole logic, including the conjunction rule, by computing the best predicate transformer corresponding to the procedure specifications used in the proof of a program (Section 5.2.1). The second is a novel proof that avoids computing the transformer at the price of being more complicated and not being able to establish the soundness of the conjunction rule (Section 5.2.2). However, unlike the former proof, this one can be adapted to concurrent setting even when resource invariants may be imprecise and the conjunction rule does not hold (Section 5.4.1). This allows us to show that our interprocedural analysis can soundly be composed with the thread-modular analysis of Chapter 3 (Section 5.4.2).

The simple treatment of procedure calls described above may not always be sufficient. Suppose we call `append` in the context

$$\mathtt{x}, \mathtt{y}, \mathtt{u}, \mathtt{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x}),$$

where `x` and `y` are global variables, and `u` and `v` are local variables of the caller. According to the above heuristic for heap splitting, the local heap should contain the data structure described by $\mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$ and the global variables `x` and `y`, and the frame should contain $\mathsf{ls}(\mathtt{v}, \mathtt{x})$ and the local variables `u` and `v`. In this splitting, we have pointers `x` and `u` between the local heap and the frame, which are called *cutpoints* [67]. If we want to know that, after `append` terminates, `x` still points into the list with the head `v`, and `u` into the one with the head `x`, our analysis has to treat cutpoints specially. From the perspective of program logics, what we need is a specification for `append` parameterised by logical variables denoting the values of the cutpoints:

$$\{\mathtt{x}, \mathtt{y} \Vdash \mathtt{x} = Z \land \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}$$
$$\mathtt{append}$$
$$\{\mathtt{x}, \mathtt{y} \Vdash \mathtt{x} = Z \land \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\}$$

This specification can then be adapted to the calling context using the rules FRAME, EXISTS, and CONSEQ as shown in Figure 5.1.[1] We can mirror this treatment in the analysis by replacing cutpoints with logical variables and storing their values in the local heap or the frame. Upon computing the effect of the procedure on the local heap, we

[1] We use these rules to replace Y and Z with the program variables `u` and `x` because our assertion language treats variables as resource [62] (the benefit being that our proof rules do not contain side conditions). In the standard Hoare logic we would just use the substitution rule [42].

$$\begin{array}{c}
\{\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\} \\
\mathtt{append} \\
\{\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})\} \\
\hline
\{(\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})) * (\mathtt{u},\mathsf{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z))\} \\
\mathtt{append} \\
(\{\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})) * (\mathtt{u},\mathsf{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z))\} \\
\hline
\{\exists Y, Z.\, (\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})) * (\mathtt{u},\mathsf{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z))\} \\
\mathtt{append} \\
\{\exists Y, Z.\, (\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})) * (\mathtt{u},\mathsf{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z))\} \\
\hline
\{\mathtt{x},\mathtt{y},\mathtt{u},\mathsf{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x})\} \\
\mathtt{append} \\
\{\mathtt{x},\mathtt{y},\mathtt{u},\mathsf{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{y}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x})\}
\end{array}$$

Figure 5.1: Adapting a procedure specification to a calling context using the rules Frame, Exists, and Conseq

can eliminate the logical values, replacing them with cutpoints. In the above example, this yields the local heap $\mathtt{x},\mathsf{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$, and the frame $\mathtt{u},\mathsf{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z)$. Note that generic summaries constructed in this way can be reused in calling contexts with the same configuration of cutpoints. For example, we can also use the above specification of `append` in the context

$$\mathtt{x},\mathtt{y},\mathtt{u},\mathtt{v},\mathtt{w} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{w}) * \mathsf{ls}(\mathtt{w}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathtt{u} \mapsto \mathtt{w} * \mathtt{v} \mapsto \mathtt{x},$$

where `u`, `v`, and `w` are local variables of the caller.

Treating cutpoints in the way described above is maximally precise. However, there is a problem that for recursive procedures it may result in the number of free logical variables denoting them growing unboundedly, causing the analysis to diverge. A solution is to abstract cutpoints beyond some bounded number using valid implications such as

$$\begin{aligned}(\mathtt{x},\mathtt{y},\mathtt{u},\mathsf{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x})) \Rightarrow& \\ (\mathtt{x},\mathtt{y},\mathtt{u},\mathsf{v} \Vdash \exists Y, Z.\, \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, Z)&).\end{aligned}$$

In this manner, we can treat bounded numbers of cutpoints. In the above example, this results in the local heap $\mathtt{x},\mathsf{y} \Vdash \exists Y.\, \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})$ and the frame $\mathtt{u},\mathsf{v} \Vdash \exists Z.\, \mathsf{ls}(\mathtt{v}, Z)$.

The initial version of the interprocedural analysis we develop in Section 5.3 abstracts away all the cutpoints. In Section 5.5, we prove the soundness of the rule Exists in our logic and construct interprocedural analyses for sequential and concurrent programs that are able to treat cutpoints precisely. As before, the proof of soundness of the logic and the analysis is complicated by the need to consider the case of imprecise resource invariants.

5.1 Programming language and semantics

Programming language. We extend the programming language of Section 2.1.3 with the parameterless procedure call command:

$$C \ ::= \ \ldots \mid \mathtt{f}$$

Programs in the extended language have the form

$$\mathtt{f}_0 \ \{\mathtt{local}\ \vec{\mathtt{x}}_0\ \mathtt{in}\ C_0\} \ \ldots \ \mathtt{f}_l \ \{\mathtt{local}\ \vec{\mathtt{x}}_l\ \mathtt{in}\ C_l\}$$

where $\mathtt{f}_0$ is the top-level procedure. We consider only well-formed programs in which the names of all procedures declared are distinct and all procedures called are declared. We also require that no procedure calls $\mathtt{f}_0$.

For simplicity of presentation our procedures do not return values and do note take parameters, as results and parameters can be passed to a procedure via specially designated global variables. For example, if the set Seq of primitive sequential commands contains the assignment command, a call $\mathtt{f}(E)$ to a procedure

$$\mathtt{f(y)}\ \{\mathtt{local}\ \vec{\mathtt{x}}\ \mathtt{in}\ C\}$$

can be encoded in our language as

$$\mathtt{z} = E;\ \mathtt{f}$$

where $\mathtt{z}$ is a fresh variable and the procedure declaration is

$$\mathtt{f}\ \{\mathtt{local}\ \vec{\mathtt{x}},\mathtt{y}\ \mathtt{in}\ \mathtt{y} = \mathtt{z};\ \mathtt{z} = \mathtt{nondet()};\ C\}$$

The command $\mathtt{z} = \mathtt{nondet()}$ assigns $\mathtt{z}$ a non-deterministically chosen value. We add it so that the analyses we construct in this chapter did not have to track the irrelevant value of the variable $\mathtt{z}$ after the procedure $\mathtt{f}$ reads it into a local variable.

Let us fix a program S of the above form. We represent every procedure body C_i, $i = 0..l$ with its CFG $(N_i, T_i, \mathsf{start}_i, \mathsf{end}_i)$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{f}_i \mid i = 0..l\}$ and let $N = \bigcup_{i=0}^{l} N_i$ and $T = \bigcup_{i=0}^{l} T_i$. Without loss of generality, we require that the CFGs be deterministic for procedure calls, i.e., for any $v \in N$ there must exist at most one procedure-call edge starting from v and at most one procedure-call edge ending at v. Thus, for an edge $(v, \mathtt{f}_i, v') \in T$ we can define $\mathsf{call}(v') = v$ and $\mathsf{calledc}(v) = \mathsf{calledr}(v') = i$. We also define $\mathsf{proc}(v) = k$ if $v \in N_k$. For a procedure-call edge $(v, \mathtt{f}_i, v')$ we call v the call point, v' the return point, start_i the starting point, and end_i the final point.

Model of program states. The logic for programs with procedures we present here is abstract: as in Chapter 3, the technical development is done with respect to a class of models of program states. Namely, we assume a separation algebra $\Sigma_{\mathbf{e}}$ representing mutable procedure environments that store local variables and an algebra Σ representing the heap and the global variables. We define a separation algebra on $\Sigma_{\mathbf{e}} \times \Sigma$ by lifting $*$ to $\Sigma_{\mathbf{e}} \times \Sigma$ componentwise. Let $D_{\mathbf{e}} = \mathcal{P}(\Sigma_{\mathbf{e}})^\top$, $D = \mathcal{P}(\Sigma)^\top$, and $D_2 = \mathcal{P}(\Sigma_{\mathbf{e}} \times \Sigma)^\top$ be the separation domains constructed out of the separation algebras $\Sigma_{\mathbf{e}}$, Σ, and $\Sigma_{\mathbf{e}} \times \Sigma$, respectively. We assume that $\Sigma_{\mathbf{e}}$ and Σ are such that $D_{\mathbf{e}}$ and D have unit elements $e_{\mathbf{e}}$ and e. Then $(e_{\mathbf{e}}, e)$ is the unit element for D_2. For $p \in D_2 \backslash \{\top\}$ let $\mathsf{env}(p) = \{\eta \mid \exists \sigma.\, (\eta, \sigma) \in p\}$, $\mathsf{state}(p) = \{\sigma \mid \exists \eta.\, (\eta, \sigma) \in p\}$, $\mathsf{Env}(p) = \mathsf{env}(p) \times e$, and $\mathsf{State}(p) = e_{\mathbf{e}} \times \mathsf{state}(p)$. We also let $\mathsf{env}(\top) = \mathsf{Env}(\top) = \mathsf{state}(\top) = \mathsf{State}(\top) = \top$. For $\sigma \in \Sigma$ let $\mathsf{lift}(\sigma) = e_{\mathbf{e}} \times \{\sigma\}$. We lift lift to D pointwise.

We assume a function $\mathsf{InitEnv}$ that for any declaration of local variables $\vec{\mathtt{x}}$ gives the set of the procedure environments $\mathsf{InitEnv}(\vec{\mathtt{x}}) \in \mathcal{P}(\Sigma_{\mathbf{e}})$ storing only these variables with arbitrary values. Let $\mathsf{Lve}_i = \mathsf{InitEnv}(\vec{\mathtt{x}}_i)$ $(i = 0..l)$ be the set of initial environments of $\mathtt{f}_i$ and $\mathsf{Lv}_i = \mathsf{Lve}_i \times e$ be the corresponding element of the domain D_2 with an empty heap.

Example model. Let us partition the set of program variables Vars into two disjoint subsets: LocalVars and GlobalVars. We let $\Sigma =$ States (Figure 2.1) with Vars replaced by GlobalVars and let $\Sigma_{\mathbf{e}} = \text{LocalVars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms})$, where the $*$ operation on $\Sigma_{\mathbf{e}}$ is defined in the same way as the $*$ operation on stacks in Section 2.1.1. We can then let $\mathsf{InitEnv}(\vec{\mathtt{x}})$ be the set of all total functions from the set of variables $\vec{\mathtt{x}}$ to Values $\times \{1\}$. Let the domain $\mathsf{RAM}_2 = \mathcal{P}(\Sigma_{\mathbf{e}} \times \Sigma)^\top$.

Program semantics. A command in the body of a procedure can access both the environment of the procedure and the heap. Correspondingly, we assume given local functions $f_C : \Sigma_{\mathbf{e}} \times \Sigma \to D_2$ defining the semantics of primitive sequential commands $C \in \mathsf{Seq}$, which we lift to predicate transformers $f_C : D_2 \to D_2$ pointwise. For example, we can construct $f_C : \Sigma_{\mathbf{e}} \times \Sigma \to D_2$ for $C \in \mathsf{SeqRAM}$ (Section 2.1.2) and the algebras $\Sigma_{\mathbf{e}}$ and Σ defined above using the following transition relation $\hookrightarrow$:

$$\begin{array}{lcl} C, (\eta, s, h, \mathbf{i}) \hookrightarrow (\eta', s', h', \mathbf{i}) & \Leftrightarrow & C, (\eta \uplus s, h, \mathbf{i}) \rightsquigarrow (\eta' \uplus s', h', \mathbf{i}) \\ C, (\eta, s, h, \mathbf{i}) \hookrightarrow \top & \Leftrightarrow & C, (\eta \uplus s, h, \mathbf{i}) \rightsquigarrow \top \end{array}$$

where the relation $\rightsquigarrow$ is defined in Figure 2.3. To this end, for $\xi \in \Sigma_{\mathbf{e}} \times \Sigma$ we let

$$f_C(\xi) = \bigsqcup \{ \{\!\!\{\xi'\}\!\!\} \mid C, \xi \hookrightarrow \xi' \}.$$

We denote with A^+ (respectively, A^*) the set of all non-empty (respectively, possibly empty) sequences of elements of the set A. In the future, we use $\cdot$ to concatenate sequences and omit it where this does not cause confusion. We also often interpret elements of A

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{(\eta, \sigma)\}) \sqsubset \top \quad (\eta', \sigma') \in f_C(\{(\eta, \sigma)\})}{c \cdot v, (\alpha \cdot \eta, \sigma) \rightarrow_S c \cdot v', (\alpha \cdot \eta', \sigma')} \tag{5.1}$$

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{(\eta, \sigma)\}) = \top}{c \cdot v, (\alpha \cdot \eta, \sigma) \rightarrow_S c \cdot v', \top} \tag{5.2}$$

$$\frac{(v, \mathtt{f}_i, v') \in T \quad \eta' \in \mathsf{Lve}_i}{c \cdot v, (\alpha, \sigma) \rightarrow_S c \cdot v' \cdot \mathsf{start}_i, (\alpha \cdot \eta', \sigma)} \tag{5.3}$$

$$\frac{}{c \cdot v \cdot \mathsf{end}_i, (\alpha \cdot \eta, \sigma) \rightarrow_S c \cdot v, (\alpha, \sigma)} \tag{5.4}$$

Figure 5.2: Operational semantics of sequential programs with procedures

as sequences of length 1. We define the semantics of the program S by the transition relation

$$\rightarrow_S \subseteq (N^+ \times (\Sigma_{\mathbf{e}}^+ \times \Sigma)) \times (N^+ \times ((\Sigma_{\mathbf{e}}^+ \times \Sigma) \cup \{\top\}))$$

in Figure 5.2, which transforms triples of

- sequences of return points for the procedures that have been called but have not yet returned ending with the program counter of the currently executing procedure (call-stacks);
- sequences of procedure environments for the procedures on the call-stack; and
- heaps.

An *initial state* of the program S is a state of the form $(\eta_0, \sigma_0) \in \Sigma_{\mathbf{e}} \times \Sigma = \Sigma_{\mathbf{e}}^1 \times \Sigma$, where $\eta_0 \in \mathsf{Lve}_0$. We say that the program S is *safe* when run from an initial state (η_0, σ_0), if it is not the case that $\mathsf{start}_0, (\eta_0, \sigma_0) \rightarrow_S^* c, \top$ for some call-stack c.

5.2 Abstract separation logic with procedures

We adapt abstract separation logic presented in Section 2.1.3 to our programming language as follows. The judgements of the new logic are of the form $\Gamma \vdash \{P\}\ C\ \{Q\}$, where Γ is a procedure context—a set of procedure specifications of the form $\{P'\}\ \mathtt{f}_i\ \{Q'\}$. Formulae of the assertion language are now interpreted with respect to $\Sigma_{\mathbf{e}} \times \Sigma$, i.e., they restrict both the current procedure environment and the heap. We assume that the assertion language contains an assertion $\mathbf{true_e}$ such that $[\![\mathbf{true_e}]\!] = \Sigma_{\mathbf{e}} \times e$. For a declaration of local variables $\vec{\mathtt{x}}$, we denote with $\mathsf{Vars}(\vec{\mathtt{x}})$ an assertion such that $[\![\mathsf{Vars}(\vec{\mathtt{x}})]\!] = \mathsf{InitEnv}(\vec{\mathtt{x}}) \times e$.

For example, the assertion language for the domain $\mathsf{RAM}_2 = \mathcal{P}(\Sigma_{\mathbf{e}} \times \Sigma)^\top$ defined in Section 5.1 is the same as for the domain RAM (Section 2.1.1). However, now its formulae are interpreted with respect to an environment $\eta \in \Sigma_{\mathbf{e}}$, as well as a stack, a heap, and an interpretation $(s, h, \mathbf{i}) \in \Sigma$:

$$(\eta, s, h, \mathbf{i}) \models P \Leftrightarrow (\eta \uplus s, h, \mathbf{i}) \models P,$$

where the later $\models$ corresponds to the notion of validity defined in Section 2.1.1. In this case $\mathsf{Vars}(\vec{\mathtt{x}})$ is $\vec{\mathtt{x}} \Vdash \mathsf{emp}_{\mathbf{h}}$. Note that the axioms in Figure 2.5 are sound for the domain RAM_2 and the corresponding transformers defined in Section 5.1.

The proof rules of separation logic (Figure 2.4) are adapted to the new setting by prefixing every triple in them with $\Gamma \vdash$. Additionally, we have the following procedure call axiom:

$$\frac{}{\Gamma, \{P\}\ \mathtt{f}_i\ \{Q\} \vdash \{P\}\ \mathtt{f}_i\ \{Q\}}\ \textsc{ProcCall}$$

At the moment we do not consider the rules Exists and Forall (see Section 5.5). A proof of the program S is given by triples

$$\Gamma \vdash \{P * \mathsf{Vars}(\vec{\mathtt{x}}_i)\}\ C_i\ \{Q * \mathbf{true_e}\} \text{ for every } \{P\}\ \mathtt{f}_i\ \{Q\} \in \Gamma. \tag{5.5}$$

Note that we place no requirements on the environment of a procedure in its postcondition, since the corresponding rule (5.4) of the operational semantics discards the environment upon return from the procedure.

We say that $p \in D_2$ *has an empty environment*, if $p = \top$ or $p \subseteq e_{\mathbf{e}} \times p'$ for some $p' \subseteq \Sigma$. We require that in the above proof the denotations of pre- and postconditions in the specifications have an empty environment. This requirement prevents a caller from passing a part of its local environment to the callee.

Our aim in the rest of this section is to prove the following theorem stating the soundness of abstract separation logic with procedures.

Theorem 5.1 (Soundness of the logic). *Given a proof (5.5) of the program S, where $\{P_0\}\ \mathtt{f}_0\ \{Q_0\} \in \Gamma$, suppose $(\eta_0, \sigma_0) \in [\![P_0 * \mathsf{Vars}(\vec{\mathtt{x}}_0)]\!]$. Then the program S is safe when run from the initial state (η_0, σ_0), and whenever $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* \mathsf{end}_0, (\eta, \sigma)$, we have $\mathsf{lift}(\sigma) \subseteq [\![Q_0]\!]$.*

As noted at the beginning of this chapter, we give two proofs of the theorem: a simpler proof that establishes the soundness of the whole logic (Section 5.2.1) and a more complicated one that establishes Theorem 5.1 under the assumption that the conjunction rule is not used in the proof of the program (Section 5.2.2). We use the latter proof to establish the soundness of our interprocedural analysis (Section 5.3) and to show that the logic and the analysis are sound when combined with the corresponding logic and analysis for static locks (Section 5.4). The technical core of the former proof is essentially the same as in the proof of soundness of proof rules for information hiding in heap-manipulating programs [58]. We give it here, first, to establish the soundness of the conjunction rule in the logic for procedures we propose and, second, to illustrate the difference between the two proofs. In particular, in Section 5.4 we explain why the standard proof does not generalise to concurrent programs in the case of imprecise resource invariants.

We make use of the following auxiliary notions. A *semantic procedure context $\mathcal{E}$* for a set of procedures $\{\mathtt{f}_0, \ldots, \mathtt{f}_l\}$ is a (possibly infinite) multiset of semantic specifications, each of the form $\{p\}$ $\mathtt{f}_i$ $\{q\}$, where $i \in \{0, \ldots, l\}$ and $p, q \in D_2$ have an empty environment. We typically represent a semantic procedure context by indexing its specifications with elements of a set $\mathcal{R}$:

$$\mathcal{E} = \big\{\{p_i^j\}\ \mathtt{f}_i\ \{q_i^j\} \mid i = 0..l,\ j \in \mathsf{specs}(i)\big\}, \tag{5.6}$$

where $\mathsf{specs}(i) \subseteq \mathcal{R}$ gives the indices of the specifications for $\mathtt{f}_i$. Note that, given a procedure context Γ used in a proof of a program, we can construct a semantic procedure context containing the semantic specification $\{[\![P]\!]\}$ $\mathtt{f}_i$ $\{[\![Q]\!]\}$ for every specification $\{P\}$ $\mathtt{f}_i$ $\{Q\}$ in Γ.

We define an operation

$$\odot : (\Sigma_{\mathsf{e}}^* \times \Sigma) \times (\Sigma_{\mathsf{e}}^* \times \Sigma) \rightharpoonup \Sigma_{\mathsf{e}}^* \times \Sigma$$

as follows:

$$(\alpha_1, \sigma_1) \odot (\alpha_2, \sigma_2) = (\alpha_1\alpha_2, \sigma_1 * \sigma_2)$$

and lift it to the operation $\odot : \mathcal{P}(\Sigma_{\mathsf{e}}^* \times \Sigma)^\top \times \mathcal{P}(\Sigma_{\mathsf{e}}^* \times \Sigma)^\top \to \mathcal{P}(\Sigma_{\mathsf{e}}^+ \times \Sigma)^\top$ in the expected way. In the following we use elements of $\Sigma_{\mathsf{e}} \times \Sigma$ as arguments of $\odot$, interpreting them as elements of $\Sigma_{\mathsf{e}}^* \times \Sigma$, where the sequence of environments is of length 1. We define the iterated version of the lifted $\odot$:

$$\bigodot_{i=1}^{k} p_i = (((\{\mathsf{e}\} \times e) \odot p_1) \odot \ldots) \odot p_k,$$

where e is the empty sequence.

5.2.1 Soundness of the logic with the conjunction rule

Consider the proof (5.5) of the program S. Let $\mathcal{E}$ be the semantic procedure context constructed out of the procedure context Γ, which we represent in the form (5.6) with $\mathcal{R} = \mathbb{N}$.

Given the specifications for a procedure $\mathtt{f}_i$ in $\mathcal{E}$, it is possible to construct the best (in the sense of [15]) local function $g_i : \Sigma \to D$ defining the meaning of the procedure consistent with the specifications, which describes how the procedure can change the global heap.[2] We define g_i as follows: for $\sigma \in \Sigma$ we let

$$g_i(\sigma) = \bigsqcap \big\{\mathsf{state}(q_i^j) * \{\sigma'\} \mid \sigma \in \mathsf{state}(p_i^j) * \{\sigma'\} \wedge \sigma' \in \Sigma \wedge j \in \mathsf{specs}(i)\big\}. \tag{5.7}$$

Informally, to compute the effect of the procedure $\mathtt{f}_i$ on a heap σ, we consider all the ways in which σ can be split into a part satisfying a precondition of $\mathtt{f}_i$ and a frame, and take the meet of the corresponding postconditions combined with the frames. Thus, we compute the smallest set of possible outcomes of executing the procedure consistent with

[2] Calcagno et al. [15] call it best local action.

the specifications. It is easy to show that the functions constructed in this way are local. Let $\vec{g}$ be the vector of g_i, $i = 0..l$. We lift g_i to D_2 pointwise.

The idea of the proof of Theorem 5.1 we present here is to show that the functions g_i constructed above over-approximate the denotational semantics of procedures in the program and then to establish a connection between g_i and the operational semantics of the program. We now define the denotational semantics of the procedures, given by a vector in $(\Sigma \to D)^{l+1}$.

Consider an approximation $\vec{f} : (\Sigma \to D)^{l+1}$ of the meaning of the procedures in the program. We first define the meaning of the commands C_i $(i = 0..l)$ in the case when the meaning of the procedures called by C_i is given by $\vec{f}$. This is defined by the function $R_i(\vec{f}) : D_2 \times N_i \to D_2$ such that $R_i(\vec{f}, p, v)$ gives the set of states reachable at the program point v when executing C_i from an initial state satisfying p. Its formal definition is similar to the fixed-point characterisation of collecting semantics in Section 2.2, with the effect of a procedure call command $\mathtt{f}_i$ determined by f_i. Namely, consider a functional $\mathcal{F}_i(\vec{f}, p) : (N_i \to D_2) \to (N_i \to D_2)$ defined as follows: $\mathcal{F}_i(\vec{f}, p)(G) = \widetilde{G}$ where $\widetilde{G}(\mathsf{start}_i) = p$ and for every program point $v' \in N_i \backslash \{\mathsf{start}_i\}$

$$\widetilde{G}(v') = \bigsqcup_{(v,C,v') \in T_i} \mathsf{Post}(C, G(v)),$$

where $\mathsf{Post}(C) = f_C$ for $C \in \mathsf{Seq}$ and $\mathsf{Post}(\mathtt{f}_i)$ is the pointwise lifting to D_2 of the corresponding function:

$$\mathsf{Post}(\mathtt{f}_i, (\eta, \sigma)) = \begin{cases} \{\eta\} \times f_i(\sigma), & \text{if } f_i(\sigma) \neq \top; \\ \top, & \text{if } f_i(\sigma) = \top. \end{cases}$$

The definition of $\mathsf{Post}(\mathtt{f}_i)$ reflects the fact that a callee may not change the local environment of the caller. We can then define $R_i(\vec{f}, p, v) = (\mathsf{lfp}(\mathcal{F}_i(\vec{f}, p)))(v)$.

The standard definition of the denotational semantics of procedures [79] is adapted to our setting as follows. Consider the functional $\mathcal{G} : (\Sigma \to D)^{l+1} \to (\Sigma \to D)^{l+1}$ defined as follows: $\mathcal{G}(\vec{f}) = \vec{f}'$, where for any $\sigma \in \Sigma$ and $i = 0..l$

$$f_i'(\sigma) = \mathsf{state}(R_i(\vec{f}, \mathsf{lift}(\sigma) * \mathsf{Lv}_i, \mathsf{end}_i)).$$

The denotational semantics $\vec{f} = \mathsf{lfp}(\mathcal{G})$ is the least fixed point of $\mathcal{G}$ under the pointwise extension of the order on D.

We now proceed to show that the functions g_i over-approximate the denotational semantics of the procedures in the program, i.e., $\vec{f} \sqsubseteq \vec{g}$. The following lemma shows the soundness of the proof (5.5) with respect to the interpretation of C_i defined by $R_i(\vec{g})$.

Lemma 5.2. $\forall i = 0..l, j \in \mathsf{specs}(i). \mathsf{State}(R_i(\vec{g}, p_i^j * \mathsf{Lv}_i, \mathsf{end}_i)) \sqsubseteq q_i^j$.

The proof is similar to that of Lemma 3.10. The soundness of the rule ProcCall follows from the fact that $g_i(\mathsf{state}(p_i^j)) \sqsubseteq \mathsf{state}(q_i^j)$. The rules Disj and Conj are sound because

g_i distribute over $\sqcup$ and $\sqcap$. The soundness of FRAME follows from the locality of g_i. In particular, we note for the future that $R_i(\vec{g}, \cdot, \mathsf{end}_i)$ is local:

$$\forall i = 0..l.\, \forall p, q \in D_2.\, R_i(\vec{g}, p * q, \mathsf{end}_i) \sqsubseteq R_i(\vec{g}, p, \mathsf{end}_i) * q. \tag{5.8}$$

We show that $\vec{f} \sqsubseteq \vec{g}$ using Park induction, i.e., by establishing that $\mathcal{G}(\vec{g}) \sqsubseteq \vec{g}$. This follows from the following lemma stating that the meaning of the body of a procedure computed using the transformers $\vec{g}$ to interpret procedure calls is at least as precise as that given by the corresponding transformer.

Lemma 5.3. $\forall i = 0..l.\, \forall \sigma \in \Sigma.\, \mathsf{state}(R_i(\vec{g}, \mathsf{lift}(\sigma) * \mathsf{Lv}_i, \mathsf{end}_i)) \sqsubseteq g_i(\sigma)$.

Proof. Consider i, j, and σ' such that

$$\sigma \in \mathsf{state}(p_i^j) * \{\sigma'\}. \tag{5.9}$$

It is sufficient to prove that $\mathsf{state}(R_i(\vec{g}, \mathsf{lift}(\sigma) * \mathsf{Lv}_i, \mathsf{end}_i)) \sqsubseteq \mathsf{state}(q_i^j) * \{\sigma'\}$. Indeed:

$$\begin{array}{rll} \mathsf{state}(R_i(\vec{g}, \mathsf{lift}(\sigma) * \mathsf{Lv}_i, \mathsf{end}_i)) & \sqsubseteq \mathsf{state}(R_i(\vec{g}, p_i^j * \mathsf{lift}(\sigma') * \mathsf{Lv}_i, \mathsf{end}_i)) & (5.9) \\ & \sqsubseteq \mathsf{state}(R_i(\vec{g}, p_i^j * \mathsf{Lv}_i, \mathsf{end}_i)) * \{\sigma'\} & (5.8) \\ & \sqsubseteq \mathsf{state}(q_i^j) * \{\sigma'\} & \text{Lemma 5.2} \end{array}$$

□

Using the above lemma, we can connect the over-approximation of the denotational semantics of procedures given by g_i to the operational semantics of Section 5.1.

Lemma 5.4. *Suppose* $\eta_0 \in \mathsf{Lve}_0$, $\sigma_0 \in \Sigma$, *and* $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$. *Then there exists a sequence* $\xi_t \in (\Sigma_{\mathsf{e}} \times \Sigma) \cup \{\top\}$ *for* $t = 0..k$ *such that*

$$\begin{array}{r} \xi_0 = (\eta_0, \sigma_0); \quad \{\!|\xi_t|\!\} \sqsubseteq R_{i_t}(\vec{g}, \mathsf{State}(\{\!|\xi_{t-1}|\!\}) * \mathsf{Lv}_{i_t}, \mathsf{call}(v_t)),\ t = 1..(k-1); \\ \{\!|\xi_k|\!\} \sqsubseteq R_{i_k}(\vec{g}, \mathsf{State}(\{\!|\xi_{k-1}|\!\}) * \mathsf{Lv}_{i_k}, v_k), \end{array} \tag{5.10}$$

where $i_t = \mathsf{proc}(v_t)$, $t = 1..k$, *and*

$$\{\!|\zeta|\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mathsf{Env}(\{\!|\xi_t|\!\}) \right) \odot \{\!|\xi_k|\!\}. \tag{5.11}$$

In the statement of the lemma, ξ_t for $t = 1..(k-1)$ define the state of the program at the points in the execution where the procedure $\mathtt{f}_{i_{t+1}}$ is called; ξ_k then gives the state at the current call-stack. We use $R_{i_{t+1}}$ to compute the effect of executing a procedure until the next procedure on the call-stack is called.

Proof of Lemma 5.4. We prove the lemma by induction on the length of the derivation of the state ζ. Here we show only the most interesting case—that of procedure return (5.4). Consider a transition in the operational semantics of the form

$$v_1 \dots v_{k-1}\mathsf{end}_{i_k}, (\eta_1 \dots \eta_{k-1}\eta_k, \sigma) \to_S v_1 \dots v_{k-1}, (\eta_1 \dots \eta_{k-1}, \sigma)$$

and assume a sequence ξ_t, $t = 0..k$ satisfying (5.10) and (5.11) for $\zeta = (\eta_1 \dots \eta_{k-1}\eta_k, \sigma)$. Suppose $\xi_t \neq \top$ for $t = 1..(k-1)$ (the converse case is trivial). Then

$$\begin{aligned}
\{(\eta_{k-1}, \sigma)\} &\sqsubseteq \mathsf{Env}(\{\xi_{k-1}\}) * \mathsf{State}(\{\!|\xi_k|\!\}) && (5.11)\\
&\sqsubseteq \mathsf{Env}(\{\xi_{k-1}\}) * \mathsf{State}(R_{i_k}(\vec{g}, \mathsf{State}(\{\xi_{k-1}\}) * \mathsf{Lv}_{i_k}, \mathsf{end}_{i_k})) && (5.10) \text{ for } \xi_k\\
&\sqsubseteq \mathsf{Env}(\{\xi_{k-1}\}) * \mathsf{lift}(g_{i_k}(\mathsf{state}(\{\xi_{k-1}\}))) && \text{Lemma 5.3}\\
&= \mathsf{Post}(\mathtt{f}_{i_k}, \{\xi_{k-1}\}) && \text{definition of } \mathsf{Post}\\
&= \mathsf{Post}(\mathtt{f}_{i_k}, R_{i_{k-1}}(\vec{g}, \mathsf{State}(\{\xi_{k-2}\}) * \mathsf{Lv}_{i_{k-1}}, \mathsf{call}(v_{k-1}))) && (5.10) \text{ for } \xi_{k-1}\\
&\sqsubseteq \mathsf{state}(R_{i_{k-1}}(\vec{g}, \mathsf{State}(\{\xi_{k-2}\}) * \mathsf{Lv}_{i_{k-1}}, v_{k-1})) && \text{definition of } R_{i_{k-1}}
\end{aligned}$$

Let $\xi'_{k-1} = (\eta_{k-1}, \sigma)$, then the sequence $\xi_0, \xi_1, \dots, \xi_{k-2}, \xi'_{k-1}$ is the required one. $\square$

Proof of Theorem 5.1. Let j_0 be the index of the specification $\{P_0\}\ \mathtt{f}_0\ \{Q_0\}$ in $\mathcal{E}$. Partial correctness follows from Lemmas 5.2 (for $i = 0$, $j = j_0$) and 5.4. Suppose now that the program S is unsafe when run from $(\eta_0, \sigma_0) \in [\![P_0 * \mathsf{Vars}(\vec{\mathtt{x}}_0)]\!]$. Then $(\eta_0, \sigma_0) \in p_0^{j_0} * \mathsf{Lv}_0^{j_0}$ and $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \top$. By Lemma 5.4, for some u we have $R_{i_u}(\vec{g}, \mathsf{State}(\xi_{u-1}) * \mathsf{Lv}_{i_u}, v_u) = \top$, hence $R_{i_k}(\vec{g}, \mathsf{State}(\xi_{k-1}) * \mathsf{Lv}_{i_k}, \mathsf{end}_{i_k}) = \top$ by the definition of R_{i_k}. We now show by induction on t that $R_{i_t}(\vec{g}, \mathsf{State}(\xi_{t-1}) * \mathsf{Lv}_{i_t}, \mathsf{end}_{i_t}) = \top$ for $t = u, u-1, \dots, 1$. Assume that $2 \leq t \leq u$ and $R_{i_t}(\vec{g}, \mathsf{State}(\xi_{t-1}) * \mathsf{Lv}_{i_t}, \mathsf{end}_{i_t}) = \top$. Then by Lemma 5.3 we have $g_{i_t}(\mathsf{state}(\xi_{t-1})) = \top$. It is easy to show that this entails $R_{i_{t-1}}(\vec{g}, \mathsf{State}(\xi_{t-2}) * \mathsf{Lv}_{i_{t-1}}, \mathsf{end}_{i_{t-1}}) = \top$, which completes the induction. Letting $t = 1$ in the induction hypothesis, we now get $R_0(\vec{g}, \mathsf{State}(\xi_0) * \mathsf{Lv}_0, \mathsf{end}_0) = \top$, which contradicts Lemma 5.2 for $i = 0$, $j = j_0$. $\square$

5.2.2 Soundness of the logic without the conjunction rule

Our second proof of soundness is done with the aid of a procedure-local interpretation of the body of each procedure, defined using semantic proofs. Given a set of specification indices $\mathcal{R}$ and a set of procedures $\{\mathtt{f}_0, \dots, \mathtt{f}_l\}$, here we define a semantic proof as a tuple $(C, G, \mathcal{E}, \mu)$, where

- C is a command with a CFG $(N, T, \mathsf{start}, \mathsf{end})$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{f}_0, \dots, \mathtt{f}_l\}$;
- $G : N \to D_2$ maps program points of C to semantic annotations;
- $\mathcal{E}$ is a semantic procedure context of the form (5.6) for the given set of indices $\mathcal{R}$;

- $\mu : N \times \mathcal{R} \to D_2$ is a mapping such that for an edge $(v, \mathtt{f}_i, v') \in T$, $\mu(v', j)$ gives the frame for the jth specification of the procedure called at v

such that for all edges $(v, C', v') \in T$

- if $C' \in \mathsf{Seq}$, then
$$f_{C'}(G(v)) \sqsubseteq G(v'); \tag{5.12}$$
- if C' is $\mathtt{f}_i$, then
$$G(v) \sqsubseteq \bigsqcup_{j \in \mathsf{specs}(i)} p_i^j * \mu(v', j) \tag{5.13}$$
and
$$G(v') \sqsupseteq \bigsqcup_{j \in \mathsf{specs}(i)} q_i^j * \mu(v', j). \tag{5.14}$$

Inequalities (5.13) and (5.14) represent a semantic counterpart of the axiom ProcCall closed under the applications of the rules Frame and Disj (so that we could establish an analogue of Lemma 3.8(i,ii) justifying the soundness of these rules). The pre- and postconditions in the inequalities are put into a normal form, where the frame for every specification is given by μ. As we show below, we can always extract a semantic proof of this form from a proof of a procedure that does not use the conjunction rule.

The following lemma establishes the soundness of a proof of the program S, given in terms of procedural-local semantic proofs with respect to its operational semantics.

Lemma 5.5 (Soundness of the procedure-local interpretation). *Consider a semantic procedure context $\mathcal{E}$ of the form (5.6) for some set of indices $\mathcal{R}$. Assume semantic proofs $(C_i, G_i^j, \mathcal{E}, \mu(i,j))$ for $i = 0..l$, $j \in \mathsf{specs}(i)$,*[3] *where $G_i^j(\mathsf{start}_i) \sqsupseteq p_i^j * \mathsf{Lv}_i$ and $\mathsf{State}(G_i^j(\mathsf{end}_i)) \sqsubseteq q_i^j$, and take $j_1 \in \mathsf{specs}(0)$. If $(\eta_0, \sigma_0) \in \mathsf{Lve}_0 \times \Sigma$ is such that*
$$\{(\eta_0, \sigma_0)\} \sqsubseteq G_0^{j_1}(\mathsf{start}_0), \tag{5.15}$$
then whenever $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$, for some $j_2, \dots, j_k$ such that $j_t \in \mathsf{specs}(i_t)$, $t = 2..k$ we have
$$\{\!|\zeta|\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot G_{i_k}^{j_k}(v_k), \tag{5.16}$$
where $i_t = \mathsf{proc}(v_t)$, $t = 1..k$.

Informally, the lemma establishes that for any state at a given call-stack there exist specifications for the procedures on the call-stack such that the state can be obtained by combining the frames for the specifications in the corresponding semantic proofs and the current procedure-local state.

[3] Thus, here μ is a function that, given indices of a procedure and its summary, produces the μ-component of the corresponding semantic proof.

Proof of Lemma 5.5. We prove the statement of the lemma by induction on the length of the derivation of ζ in the operational semantics of the program S. In the base case (5.16) is equivalent to (5.15). Suppose now that $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$, and (5.16) holds. We consider three cases corresponding to the next rule of the operational semantics applied in the computation.

Case 1. Rules (5.1) and (5.2): primitive sequential command. We have

$$v_1 \dots v_{k-1} v_k, (\eta_1 \dots \eta_{k-1} \eta_k, \sigma) \to_S v_1 \dots v_{k-1} v'_k, \zeta'$$

and $(v_k, C, v'_k) \in T$, $C \in \mathsf{Seq}$. It is sufficient to show that

$$\{\!\![\zeta']\!\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot G_{i_k}^{j_k}(v'_k). \tag{5.17}$$

This holds trivially if $\mu(i_t, j_t, v_t, j_{t+1}) = \top$ for some t. Otherwise, from (5.16) it follows that

$$\{(\eta_k, \sigma)\} \sqsubseteq G_{i_k}^{j_k}(v_k) * \mathsf{lift}(\sigma'') \tag{5.18}$$

for some $\sigma'' \in \Sigma$ such that $(\eta_1 \dots \eta_{k-1}, \sigma'') \in \bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1})$. Then

$$\begin{array}{lll} f_C(\{(\eta_k, \sigma)\}) & \sqsubseteq f_C(G_{i_k}^{j_k}(v_k) * \mathsf{lift}(\sigma'')) & (5.18) \\ & \sqsubseteq f_C(G_{i_k}^{j_k}(v_k)) * \mathsf{lift}(\sigma'') & f_C \text{ is local} \\ & \sqsubseteq G_{i_k}^{j_k}(v'_k) * \mathsf{lift}(\sigma'') & (5.12) \end{array}$$

If $\zeta' = \top$, then $f_C(\{(\eta_k, \sigma)\}) = \top$. From the above it then follows that $G_{i_k}^{j_k}(v'_k) = \top$, which entails (5.17). If $\zeta' \neq \top$, then $\zeta' = (\eta_1 \dots \eta_{k-1} \eta'_k, \sigma')$ and $\{(\eta'_k, \sigma'_k)\} \sqsubseteq f_C(\{(\eta_k, \sigma)\})$. Together with the above, this again entails (5.17).

Case 2. Rule (5.3): procedure call. In this case

$$v_1 \dots v_{k-1} v_k, (\eta_1 \dots \eta_{k-1} \eta_k, \sigma) \to_S v_1 \dots v_{k-1} v'_k \mathsf{start}_{i_{k+1}}, (\eta_1 \dots \eta_{k-1} \eta_k \eta_{k+1}, \sigma),$$

$\eta_{k+1} \in \mathsf{Lve}_{i_{k+1}}$, and $(v_k, \mathtt{f}_{i_{k+1}}, v'_k) \in T$. It is sufficient to show that for some $j_{k+1} \in \mathsf{specs}(i_{k+1})$ we have

$$\{(\eta_1 \dots \eta_k \eta_{k+1}, \sigma)\} \sqsubseteq \left(\left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot \mu(i_k, j_k, v'_k, j_{k+1}) \right) \odot G_{i_{k+1}}^{j_{k+1}}(\mathsf{start}_{i_{k+1}}). \tag{5.19}$$

By (5.13), we have

$$G_{i_k}^{j_k}(v_k) \sqsubseteq \bigsqcup_{t \in \mathsf{specs}(i_{k+1})} (p_{i_{k+1}}^t * \mu(i_k, j_k, v'_k, t)).$$

From this and (5.16), we get

$$\{(\eta_1 \dots \eta_k, \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot \left(\bigsqcup_{t \in \mathsf{specs}(i_{k+1})} (p_{i_{k+1}}^t * \mu(i_k, j_k, v'_k, t)) \right).$$

Thus, there exists $j_{k+1} \in \mathsf{specs}(i_{k+1})$ such that

$$\{(\eta_1 \dots \eta_k, \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1})\right) \odot (p_{i_{k+1}}^{j_{k+1}} * \mu(i_k, j_k, v'_k, j_{k+1})).$$

Since $G_{i_{k+1}}^{j_{k+1}}(\mathsf{start}_{i_{k+1}}) \sqsupseteq p_{i_{k+1}}^{j_{k+1}} * \mathsf{Lv}_{i_{k+1}}$, $p_{i_{k+1}}^{j_{k+1}}$ has an empty environment, and $\eta_{k+1} \in \mathsf{Lve}_{i_{k+1}}$, this entails (5.19).

Case 3. Rule (5.4): procedure return. In this case $k \geq 2$, $v_k = \mathsf{end}_{i_k}$, and

$$v_1 \dots v_{k-1}\mathsf{end}_{i_k}, (\eta_1 \dots \eta_{k-1}\eta_k, \sigma) \rightarrow_S v_1 \dots v_{k-1}, (\eta_1 \dots \eta_{k-1}, \sigma).$$

It is sufficient to show that

$$\{(\eta_1 \dots \eta_{k-1}, \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-2} \mu(i_t, j_t, v_t, j_{t+1})\right) \odot G_{i_{k-1}}^{j_{k-1}}(v_{k-1}). \tag{5.20}$$

Since $k \geq 2$, from (5.16) we get

$$\{(\eta_1 \dots \eta_{k-1}\eta_k, \sigma)\} \sqsubseteq \left(\left(\bigodot_{t=1}^{k-2} \mu(i_t, j_t, v_t, j_{t+1})\right) \odot \mu(i_{k-1}, j_{k-1}, v_{k-1}, j_k)\right) \odot G_{i_k}^{j_k}(\mathsf{end}_{i_k}).$$

Then

$$\{(\eta_1 \dots \eta_{k-1}, \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-2} \mu(i_t, j_t, v_t, j_{t+1})\right) \odot (\mu(i_{k-1}, j_{k-1}, v_{k-1}, j_k) * (\mathsf{State}(G_{i_k}^{j_k}(\mathsf{end}_{i_k})))).$$

Since $\mathsf{State}(G_i^j(\mathsf{end}_i)) \sqsubseteq q_i^j$, this entails

$$\{(\eta_1 \dots \eta_{k-1}, \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-2} \mu(i_t, j_t, v_t, j_{t+1})\right) \odot (\mu(i_{k-1}, j_{k-1}, v_{k-1}, j_k) * q_{i_k}^{j_k}).$$

By (5.14), we have

$$\mu(i_{k-1}, j_{k-1}, v_{k-1}, j_k) * q_{i_k}^{j_k} \sqsubseteq G_{i_{k-1}}^{j_{k-1}}(v_{k-1}).$$

The last two inequalities entail (5.20). □

Proof of Theorem 5.1. Consider a proof (5.5) of the program S. Let $\mathcal{E}$ be the semantic procedure context constructed out of the procedure context Γ, which we represent in the form (5.6) with $\mathcal{R} = \mathbb{N}$. We can show that from a derivation of a triple $\Gamma \vdash \{P * \mathsf{Vars}(\vec{\mathsf{x}}_i)\}\ C_i\ \{Q * \mathbf{true_e}\}$ that does not use CONJ we can construct a semantic proof $(C_i, G, \mathcal{E}, \mu)$ for some G and μ such that $G(\mathsf{start}_i) = [\![P]\!] * \mathsf{Lv}_i$, $\mathsf{State}(G(\mathsf{end}_i)) \sqsubseteq [\![Q]\!]$, $\forall v.\, G(v) \sqsubset \top$, and $\forall v, j.\, \mu(v, j) \sqsubset \top$. This is done in the same way as in the proof of Lemma 3.4, in particular, the soundness of the rules FRAME and DISJ is shown by establishing an analogue of Lemma 3.8(i,ii).[4] The set of semantic proofs constructed in this way from (5.5) satisfies the conditions of Lemma 5.5. Theorem 5.1 is then an easy consequence of this lemma. □

[4]Note that we cannot establish the soundness of CONJ in this way as Lemma 3.8(iii) does not hold for procedure-local semantic proofs.

5.3 Interprocedural analysis

To perform the interprocedural analysis using the treatment of procedure calls and returns proposed at the beginning of this chapter, we adapt the Reps-Horwitz-Sagiv (RHS) algorithm [64, 69] to use localisation and arbitrary abstract separation domains. We call the analysis presented here the local RHS analysis.

Analysis formulation. We assume an intraprocedural analysis defined by

- a separation algebra $(D_2^\sharp, *^\sharp)$ of abstract states with a total $*^\sharp$ operation and a unit element $e^\sharp$;
- a concretisation function $\gamma : D_2^\sharp \to D_2$;
- abstract transfer functions $f_C^\sharp : D_2^\sharp \to \mathcal{P}(D_2^\sharp)^\top$ for $C \in \mathsf{Seq}$

such that

- abstract transfer functions over-approximate the concrete ones:
$$\forall p \in D_2^\sharp.\, f_C(\gamma(p)) \sqsubseteq \gamma(f_C^\sharp(p));$$
- the abstract operation of separate combination over-approximates the concrete one:
$$\forall p, q \in D_2^\sharp.\, \gamma(p) * \gamma(q) \sqsubseteq \gamma(p *^\sharp q).$$

The interprocedural analysis we present here uses the domain of sets of elements of $D^\sharp$. Therefore, out of the above we construct

- an abstract separation domain $(\mathcal{P}(D_2^\sharp)^\top, \sqsubseteq, \bigsqcup, \bot, \top, *^\sharp, \{e^\sharp\})$;
- a monotone concretisation function $\gamma : \mathcal{P}(D_2^\sharp)^\top \to D_2$;
- abstract transfer functions $f_C^\sharp : \mathcal{P}(D_2^\sharp)^\top \to \mathcal{P}(D_2^\sharp)^\top$ for $C \in \mathsf{Seq}$

such that
$$\forall p \in \mathcal{P}(D_2^\sharp)^\top.\, f_C(\gamma(p)) \sqsubseteq \gamma(f_C^\sharp(p)) \tag{5.21}$$
and
$$\forall p, q \in \mathcal{P}(D_2^\sharp)^\top.\, \gamma(p) * \gamma(q) \sqsubseteq \gamma(p *^\sharp q). \tag{5.22}$$
We thus obtain an abstract interpretation with state separation with the abstract domain $\mathcal{P}(D_2^\sharp)^\top$ and the concrete domain D_2.

For a set $A \in \mathcal{P}(D_2^\sharp \cup \{\top\})$ let the function $\mathsf{id}(A) : (D_2^\sharp \cup \{\top\}) \to \mathcal{P}(D_2^\sharp)$ be defined as follows: $\mathsf{id}(A)(p) = \{\!|p|\!\}$ if $p \in A$ and $\mathsf{id}(A)(p) = \emptyset$ otherwise. Let $\mathsf{pre} \in D_2^\sharp$ be an abstract element representing the set of initial states of the program S such that $\gamma(\mathsf{pre})$ has an empty environment. The local RHS analysis operates on the domain $\widehat{D}^\sharp = N \to ((D_2^\sharp \cup \{\top\}) \to \mathcal{P}(D_2^\sharp)^\top)$ and is defined by the functional $\mathcal{F}^\sharp(\mathsf{pre}) : \widehat{D}^\sharp \to \widehat{D}^\sharp$ in Figure 5.3. Its definition uses the following ingredients:

$\mathcal{F}^\sharp(\mathsf{pre})(G^\sharp) = \widetilde{G}^\sharp$, where

- $\widetilde{G}^\sharp(\mathsf{start}_0) = \mathsf{id}(\{\mathsf{pre} *^\sharp \mathsf{Lv}_0^\sharp\})$;
- $\widetilde{G}^\sharp(v', p) = \bigsqcup_{(v,C,v') \in T} f_C^\sharp(G^\sharp(v,p))$

  for every program point $v' \in N$ that is not a starting or return point;
- $\widetilde{G}^\sharp(v') = \mathsf{id}\Big(\{\mathsf{ProcLocal}_i(q) *^\sharp \mathsf{Lv}_i^\sharp \mid i = \mathsf{calledc}(v) \wedge p \in D_2^\sharp \cup \{\top\} \wedge$
  $((G^\sharp(v,p) \sqsubset \top \wedge q \in G^\sharp(v,p)) \vee (G^\sharp(v,p) = \top \wedge q = \top))\}\Big)$

  for every starting point $v' \in N \backslash \{\mathsf{start}_0\}$, where $i = \mathsf{proc}(v')$.
- $\widetilde{G}^\sharp(v', p) = \bigsqcup\{\mathsf{State}^\sharp(G^\sharp(\mathsf{end}_i, \mathsf{ProcLocal}_i(q) *^\sharp \mathsf{Lv}_i^\sharp)) *^\sharp \{\!\{\mathsf{Frame}_i(q)\}\!\} \mid$
  $(G^\sharp(\mathsf{call}(v'),p) \sqsubset \top \wedge q \in G^\sharp(\mathsf{call}(v'),p)) \vee (G^\sharp(\mathsf{call}(v'),p) = \top \wedge q = \top)\}$

  for every return point $v' \in N$, where $i = \mathsf{calledr}(v')$.

Figure 5.3: Local RHS analysis

- The abstract state $\mathsf{Lv}_i^\sharp \in D_2^\sharp$, $i = 0..l$ represents the initial environment of the procedure `f_i`:
$$\mathsf{Lv}_i \sqsubseteq \gamma(\mathsf{Lv}_i^\sharp). \tag{5.23}$$
- The functions $\mathsf{ProcLocal}_i : D_2^\sharp \to D_2^\sharp$ and $\mathsf{Frame}_i : D_2^\sharp \to D_2^\sharp$ for $i = 0..l$ determine the splitting of the abstract state at a call point of the procedure $\mathtt{f}_i$: $\mathsf{ProcLocal}_i$ determines the part of the state that is passed to the callee and Frame_i the part that stays with the caller. We require that they split the state soundly:
$$\forall p \in D_2^\sharp.\, \gamma(p) \sqsubseteq \gamma(\mathsf{ProcLocal}_i(p)) * \gamma(\mathsf{Frame}_i(p)). \tag{5.24}$$
  We also require that $\gamma(\mathsf{ProcLocal}_i(p))$ have an empty environment, since the ownership of the local variables of the callee should not be transferred to the caller. We lift $\mathsf{ProcLocal}_i$ and Frame_i to $\mathcal{P}(D_2^\sharp)^\top$ pointwise.
- The function $\mathsf{State}^\sharp : D_2^\sharp \to D_2^\sharp$ projects out the environment from the states in its argument:
$$\forall p \in D_2^\sharp.\, \mathsf{State}(\gamma(p)) \sqsubseteq \gamma(\mathsf{State}^\sharp(p)). \tag{5.25}$$
  We lift $\mathsf{State}^\sharp$ to $\mathcal{P}(D_2^\sharp)^\top$ pointwise.

The analysis tabulates a function $G^\sharp \in \widehat{D}^\sharp$. Intuitively, for a state $p_1 \in D_2^\sharp \cup \{\top\}$ at the starting point of the procedure containing the program point v, $G^\sharp(v, p_1) \in \mathcal{P}(D_2^\sharp)^\top$ represents the abstract states $p_2 \in D_2^\sharp \cup \{\top\}$ at the program point v such that there exists an execution of a sequence of program statements between these two points transforming p_1 to p_2. In the original RHS algorithm every such pair of states (p_1, p_2) is called a path edge. In the case when v is a final point, it is called a summary edge. The set of all summary edges for a procedure starting from a given p_1 forms a summary of the procedure.

The treatment of ordinary program points in the definition of $\mathcal{F}^\sharp(\mathsf{pre})$ is standard, except we compute transfer functions on states at a program point v separately for every input state to the procedure containing v. The equation for a starting point propagates the local parts of states at call points of the corresponding procedure (for any input state to the caller) to starting points. The states at return points are obtained by considering every state at the corresponding call point and by $*^\sharp$-conjoining the postcondition for the summary edge starting from the local part of the state to the frame. A computation of a fixed point of $\mathcal{F}^\sharp(\mathsf{pre})$ would iteratively create new path and summary edges, analysing each procedure only once for a given local heap.

A heuristic for determining heap splittings. As we noted at the beginning of this chapter, one possible way to to split the heap at a call point of a procedure is to send to the procedure all of the heap reachable from the global variables, including those holding the values of the actual parameters [67].[5] The definition of $\mathsf{ProcLocal}_k$ and Frame_k according to this heuristic is similar to the definition of $\mathsf{Protected}_k$ and $\mathsf{ThreadLocal}_k$ for the thread-modular analysis given in Section 3.3.2. We illustrate it using the example of the domain SLL of Section 2.2.1.

Consider a variation of the domain where the assertions are interpreted with respect to the example domain RAM_2 of Section 5.1. Let $D_2^\sharp$ be the set of symbolic heaps. For a variable ownership assertion O let $\mathsf{Globals}(O)$, respectively, $\mathsf{Locals}(O)$ be the set of global, respectively, local variables in O. The functions $\mathsf{ProcLocal}_k$ and Frame_k are defined by modifying equations (3.4) and (3.5) for $\mathsf{Protected}_k$ and $\mathsf{ThreadLocal}_k$, respectively. Consider a symbolic heap $O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}$. As before, we require that the variables in S be chosen so that for each equivalence class generated by the equalities in P at most one variable from this equivalence class is present in S, with preference given to unquantified variables over quantified ones, and to global variables over others. We then let

$$\begin{aligned}\mathsf{ProcLocal}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) = \mathsf{can}(&\mathsf{Globals}(O) \Vdash \exists \vec{X}, \vec{Y}.\, (\mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Globals}(O)))\\ &[\vec{Y}/(\mathsf{vars}(\mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Globals}(O))) \cap \mathsf{Locals}(O))]) \quad (5.26)\end{aligned}$$

and

$$\begin{aligned}&\mathsf{Frame}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) =\\ &\qquad \mathsf{can}(\mathsf{Locals}(O) \Vdash \exists \vec{X}, \vec{Z}.\, (\mathsf{P} \wedge \mathsf{Unreach}(\mathsf{S}, \mathsf{Globals}(O)))[\vec{Z}/\mathsf{Globals}(O)]) \quad (5.27)\end{aligned}$$

[5] Actually, an analysis using this heuristic does not compute reachability from the global variables holding the actual parameters of procedures other than the one called, since in our desugaring of parameter passing (Section 5.1) we clear them immediately after use.

for fresh $\vec{Y}, \vec{Z}$. It is easy to check that $\mathsf{ProcLocal}_k$ and Frame_k defined in this way satisfy (5.24). We also let $\mathsf{Lv}^\sharp_k = (\vec{\mathtt{x}}_k \Vdash \mathsf{emp}_{\mathbf{h}})$ and

$$\mathsf{State}^\sharp(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) = \mathsf{can}(\mathsf{Globals}(O) \Vdash \exists \vec{X}, \vec{Y}.\, (\mathsf{P} \wedge \mathsf{S})[\vec{Y}/\mathsf{Locals}(O)])$$

for fresh $\vec{Y}$.

The above definition of heap splitting abstracts away all the cutpoints (in Section 5.5.2 we define an analysis that treats cutpoints precisely). For example, consider the symbolic heap

$$P = (\mathtt{x}, \mathtt{y}, \mathtt{u}, \mathtt{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x})),$$

where `x` and `y` are global and `u` and `v` are local. In this case

$$\mathsf{ProcLocal}_k(P) = \mathsf{can}(\mathtt{x}, \mathtt{y} \Vdash \exists Y.\, \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})) = \\ (\mathtt{x}, \mathtt{y} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}))$$

and $\mathsf{Frame}_k(P) = (\mathtt{u}, \mathtt{v} \Vdash \exists Z.\, \mathsf{ls}(\mathtt{v}, Z))$. Thus, the splitting breaks the pointers `x` and `u` between the local heap and the frame.

Soundness. We prove the soundness of the analysis by showing that its results generate an instance of the procedure-local interpretation introduced in Section 5.2.2: the set of summary edges starting from a given abstract state form a procedure specification and the set of path edges starting from the same state yield a corresponding semantic proof. Let $G^\sharp \in \widehat{D}^\sharp$ be a fixed point of $\mathcal{F}^\sharp(\mathsf{pre})$ and let $\mathcal{R} = D^\sharp_2 \cup \{\top\}$. We define $\mathsf{specs}(i)$ as the set of local abstract states without initial procedure environments that are propagated to the starting point of the procedure $\mathtt{f}_i$: $\mathsf{specs}(0) = \{\mathsf{pre}\}$ and for $i = 1..l$

$$\mathsf{specs}(i) = \big\{\mathsf{ProcLocal}_i(q) \mid i = \mathsf{calledc}(v) \wedge p \in D^\sharp_2 \cup \{\top\} \wedge \\ ((G^\sharp(v,p) \sqsubset \top \wedge q \in G^\sharp(v,p)) \vee (G^\sharp(v,p) = \top \wedge q = \top))\big\}.$$

We then define $p^r_i = \gamma(r)$ and $q^r_i = \mathsf{State}(\gamma(G^\sharp(\mathsf{end}_i, r {*^\sharp} \mathsf{Lv}^\sharp_i)))\}$ for $i = 0..l$ and $r \in \mathsf{specs}(i)$. Since p^r_i and q^r_i have an empty environment, the analysis thus constructs a semantic procedure context $\mathcal{E}$ of the form (5.6), where the specifications are indexed by the abstract states used to obtain their preconditions. For $i = 0..l$ and $r \in \mathsf{specs}(i)$ let $G^r_i : N_i \to D_2$ and $\mu(i,r) : N_i \times \mathcal{R} \to D_2$ be defined as follows:

$$G^r_i(v) = \gamma(G^\sharp(v, r *^\sharp \mathsf{Lv}^\sharp_i))$$

and if v is a return point, then

$$\mu(i,r,v,r') = \bigsqcup \big\{\gamma(\mathsf{Frame}_t(w)) \mid \{\!\!\{w\}\!\!\} \sqsubseteq G^\sharp(\mathsf{call}(v), r *^\sharp \mathsf{Lv}^\sharp_i) \wedge r' = \mathsf{ProcLocal}_t(w)\big\},$$

where $t = \mathsf{calledr}(v)$; otherwise, $\mu(i,r,v,r')$ is assigned an arbitrary value. Using (5.21)–(5.25), it is easy to show that $(C_i, G^r_i, \mathcal{E}, \mu(i,r))$, $i = 0..l$, $r \in \mathsf{specs}(i)$ are semantic proofs satisfying the conditions of Lemma 5.5, i.e., $G_i^r(\mathsf{start}_i) \sqsupseteq p_i^r * \mathsf{Lv}_i$ and $\mathsf{State}(G_i^r(\mathsf{end}_i)) \sqsubseteq q_i^r$. From this lemma we can derive a soundness statement for the local RHS analysis as follows.

Assume $(\eta_0, \sigma_0) \in \Sigma_{\mathbf{e}} \times \Sigma$ is such that $\{(\eta_0, \sigma_0)\} \sqsubseteq \gamma(\mathsf{pre}) * \mathsf{Lv}_0$ and $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$. Then by Lemma 5.5 for some $r_1, \dots, r_k$ such that $r_t \in \mathsf{specs}(i_t)$, $i_t = \mathsf{proc}(v_t)$, $t = 1..k$, we have

$$\{\!|\zeta|\!\} \sqsubseteq \bigodot_{t=1}^{k-1} \bigsqcup \{\gamma(\mathsf{Frame}_{i_{t+1}}(w)) \mid \{\!|w|\!\} \sqsubseteq G^\sharp(\mathsf{call}(v_t), r_t *^\sharp \mathsf{Lv}_{i_t}^\sharp) \wedge$$
$$r_{t+1} = \mathsf{ProcLocal}_{i_{t+1}}(w)\} \odot \gamma(G^\sharp(v_k, r_k *^\sharp \mathsf{Lv}_{i_k}^\sharp)).$$

The inequality holds if and only if there exist $w_1, \dots, w_k \in D^\sharp \cup \{\top\}$ such that

$$\{\!|w_t|\!\} \sqsubseteq G^\sharp(\mathsf{call}(v_t), r_t *^\sharp \mathsf{Lv}_{i_t}^\sharp),\ t = 1..(k-1); \quad \{\!|w_k|\!\} \sqsubseteq G^\sharp(v_k, r_k *^\sharp \mathsf{Lv}_{i_k}^\sharp);$$
$$r_{t+1} = \mathsf{ProcLocal}_{i_{t+1}}(w_t),\ t = 1..(k-1)$$

and

$$\{\!|\zeta|\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \gamma(\mathsf{Frame}_{i_{t+1}}(w_t))\right) \odot \gamma(G^\sharp(v_k, r_k *^\sharp \mathsf{Lv}_{i_k}^\sharp)).$$

Eliminating r_t, we get that for some $w_1, \dots, w_k \in D_2^\sharp \cup \{\top\}$ such that

$$\{\!|w_1|\!\} \sqsubseteq G^\sharp(v_1, \mathsf{pre} *^\sharp \mathsf{Lv}_{i_1}^\sharp); \quad \{\!|w_t|\!\} \sqsubseteq G^\sharp(\mathsf{call}(v_t), \mathsf{ProcLocal}_{i_t}(w_{t-1}) *^\sharp \mathsf{Lv}_{i_t}^\sharp),$$
$$t = 2..(k-1); \quad \{\!|w_k|\!\} \sqsubseteq G^\sharp(v_k, \mathsf{ProcLocal}_{i_k}(w_{k-1}) *^\sharp \mathsf{Lv}_{i_k}^\sharp), \tag{5.28}$$

we have

$$\{\!|\zeta|\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \gamma(\mathsf{Frame}_{i_{t+1}}(w_t))\right) \odot \gamma(w_k). \tag{5.29}$$

The abstract states w_t for $t = 1..(k-1)$ give the local states for the procedures on the call-stack at the point where the procedure $\mathtt{f}_{i_t}$ is called; w_k gives the local state at the current point in the last procedure called. We have thus proved the following theorem stating that for any concrete state reachable in the operational semantics of the program, there exists an interprocedural path through the summary table computed by the analysis such that the frames on the path together with the local state of the last procedure called over-approximate the concrete state.

Theorem 5.6 (Soundness of the analysis). *Let $G^\sharp \in \widehat{D}^\sharp$ be a fixed point of $\mathcal{F}^\sharp(\mathsf{pre})$. If $(\eta_0, \sigma_0) \in \Sigma_{\mathbf{e}} \times \Sigma$ is such that $\{(\eta_0, \sigma_0)\} \sqsubseteq \gamma(\mathsf{pre}) * \mathsf{Lv}_0$, then whenever $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$, for some $w_1, \dots, w_k \in D_2^\sharp \cup \{\top\}$ satisfying (5.28) we have (5.29).*

We note that the soundness statement given by this theorem can be easily generalised from providing an over-approximation of the collecting semantics of the program to providing an over-approximation of its trace semantics. Such a statement could be used, for example, to enable compiler optimisations across procedure boundaries.

5.4 Concurrent programs with procedures

We now show how the logic and the analysis presented in this chapter can be composed with a logic and an analysis for concurrent programs. For technical simplicity we consider the logic and the analysis for static locks presented in Chapter 3. The case of storable locks is handled in a similar way.

Consider a concurrent program with procedures S of the form

$$\mathtt{f}_0\ \{\mathtt{local}\ \vec{\mathtt{x}}_0\ \mathtt{in}\ C_0\}\ \ldots\ \mathtt{f}_l\ \{\mathtt{local}\ \vec{\mathtt{x}}_l\ \mathtt{in}\ C_l\}\ \mathtt{in}\ \mathtt{f}_{\mathsf{main}1} \parallel \ldots \parallel \mathtt{f}_{\mathsf{main}n}$$

Here $\mathtt{f}_{\mathsf{main}1}, \ldots, \mathtt{f}_{\mathsf{main}n}$, $0 \leq \mathsf{main}t \leq l$, $t = 1..n$ are top-level procedures in the corresponding threads. Commands are in the language of Section 2.1.3 extended with the procedure call command as well as critical regions over the locks $\ell_1, \ldots, \ell_m$:

$$C \quad ::= \quad \ldots \mid \mathtt{f} \mid \mathtt{acquire}(\ell_k); C; \mathtt{release}(\ell_k)$$

We assume that no procedure calls the top-level procedures.

We represent the body C_i of every procedure $\mathtt{f}_i$ with its CFG $(N_i, T_i, \mathsf{start}_i, \mathsf{end}_i)$ over the set of primitive commands

$$\mathsf{Seq} \cup \{\mathtt{f}_i \mid i = 0..l\} \cup \{\mathtt{acquire}(\ell_k) \mid k = 1..m\} \cup \{\mathtt{release}(\ell_k) \mid k = 1..m\}$$

and let $N = \bigcup_{i=0}^{l} N_i$ and $T = \bigcup_{i=0}^{l} T_i$. In the following we use Lve_i, Lv_i, proc, call, $\mathsf{calledc}$, $\mathsf{calledr}$ as defined in Section 5.1.

We define the semantics of the program S by the transition relation

$$\begin{aligned}\rightarrow_S \ \subseteq \ & ((\{1, \ldots, n\} \rightarrow N^+) \times ((\{1, \ldots, n\} \rightarrow \Sigma_{\mathsf{e}}^+) \times \Sigma)) \times \\ & ((\{1, \ldots, n\} \rightarrow N^+) \times (((\{1, \ldots, n\} \rightarrow \Sigma_{\mathsf{e}}^+) \times \Sigma) \cup \{\top\}))\end{aligned}$$

in Figure 5.4, which transforms triples of

- mappings from thread identifiers to call-stacks (program counters);
- mappings from thread identifiers to sequences of environments on the call-stack of the corresponding thread; and
- heaps.

The semantics is a straightforward combination of the ones in Figures 5.2 and 3.2. As in Figure 3.2, we rely on the fact that we can determine the set $\mathsf{Free}(\mathsf{pc})$ of indices of free locks at every program counter $\mathsf{pc} \in \{1, \ldots, n\} \rightarrow N^+$.

Let us denote with pc_0 the initial program counter $\mathsf{pc}_0 = [1 : \mathsf{start}_{\mathsf{main}1}] \ldots [n : \mathsf{start}_{\mathsf{main}n}]$ and with pc_{f} the final one $[1 : \mathsf{end}_{\mathsf{main}1}] \ldots [n : \mathsf{end}_{\mathsf{main}n}]$. An *initial state* of the program S is a state of the form $([1 : \eta_1] \ldots [n : \eta_n], \sigma_0)$, where $\sigma_0 \in \Sigma$ and $\eta_k \in \mathsf{Lve}_{\mathsf{main}k}$, $k = 1..n$. We say that the program S is *safe* when run from an initial state (g_0, σ_0), if it is not the case that $\mathsf{pc}_0, (g_0, \sigma_0) \rightarrow_S^* \mathsf{pc}, \top$ for some program counter pc.

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{(\eta, \sigma)\}) \sqsubset \top \quad (\eta', \sigma') \in f_C(\{(\eta, \sigma)\})}{\mathsf{pc}[k : c \cdot v], (g[k : \alpha \cdot \eta], \sigma) \to_S \mathsf{pc}[k : c \cdot v'], (g[k : \alpha \cdot \eta'], \sigma')}$$

$$\frac{(v, C, v') \in T \quad C \in \mathsf{Seq} \quad f_C(\{(\eta, \sigma)\}) = \top}{\mathsf{pc}[k : c \cdot v], (g[k : \alpha \cdot \eta], \sigma) \to_S \mathsf{pc}[k : c \cdot v'], \top}$$

$$\frac{(v, \mathtt{acquire}(\ell_j), v') \in T \quad j \in \mathsf{Free}(\mathsf{pc}[k : c \cdot v])}{\mathsf{pc}[k : c \cdot v], (g[k : \alpha \cdot \eta], \sigma) \to_S \mathsf{pc}[k : c \cdot v'], (g[k : \alpha \cdot \eta], \sigma)}$$

$$\frac{(v, \mathtt{release}(\ell_j), v') \in T}{\mathsf{pc}[k : c \cdot v], (g[k : \alpha \cdot \eta], \sigma) \to_S \mathsf{pc}[k : c \cdot v'], (g[k : \alpha \cdot \eta], \sigma)}$$

$$\frac{(v, \mathtt{f}_i, v') \in T \quad \eta' \in \mathsf{Lve}_i}{\mathsf{pc}[k : c \cdot v], (g[k : \alpha], \sigma) \to_S \mathsf{pc}[k : c \cdot v' \cdot \mathsf{start}_i], (g[k : \alpha \cdot \eta'], \sigma)}$$

$$\frac{}{\mathsf{pc}[k : c \cdot v \cdot \mathsf{end}_i], (g[k : \alpha \cdot \eta], \sigma) \to_S \mathsf{pc}[k : c \cdot v], (g[k : \alpha], \sigma)}$$

Figure 5.4: Operational semantics of concurrent programs with procedures

5.4.1 Logic

The logic for concurrent programs with procedures is obtained by combining the logics in Sections 5.2 and 2.1.4. Its judgements are of the form $\Gamma, I \vdash \{P\}\ C\ \{Q\}$ with assertions interpreted over the domain D_2, and the proof rules are adapted straightforwardly. A proof of the program S is given by triples

$$\Gamma, I \vdash \{P * \mathsf{Vars}(\vec{\mathtt{x}}_i)\}\ C_i\ \{Q * \mathbf{true}_{\mathsf{e}}\} \text{ for every } \{P\}\ \mathtt{f}_i\ \{Q\} \in \Gamma. \tag{5.30}$$

As before, we require that pre- and postconditions of the specifications in Γ have an empty environment. Additionally, we require the same of resource invariants, since environments are local to procedure invocations and should not be transferred between threads.

The soundness statement for the logic generalises Theorems 5.1 and 3.4.

Theorem 5.7 (Soundness of the logic). *Consider a proof (5.30) of the program S, where $\{P_t\}\ \mathtt{f}_{\mathsf{main}t}\ \{Q_t\} \in \Gamma$, $t = 1..n$ and either*

- *the resource invariants in I are precise and the $*$ operation is cancellative; or*
- Conj *is not used in the derivation of the triples.*

Assume $g_0 \in \{1, \ldots, n\} \to \Sigma_{\mathsf{e}}$ is such that $g_0(t) \in \mathsf{Lve}_{\mathsf{main}t}$ for $t = 1..n$, and $\sigma_0 \in \Sigma$ is such that

$$\mathsf{lift}(\sigma_0) \subseteq \left(\underset{t=1}{\overset{n}{\circledast}} [\![P_t]\!] \right) * \left(\underset{t=1}{\overset{m}{\circledast}} [\![I_t]\!] \right).$$

Then the program S is safe when run from the initial state (g_0, σ_0), and whenever $\mathsf{pc}_0, (g_0, \sigma_0) \to_S^* \mathsf{pc}_{\mathsf{f}}, (g, \sigma)$, we have

$$\mathsf{lift}(\sigma) \subseteq \left(\underset{t=1}{\overset{n}{\circledast}} [\![Q_t]\!] \right) * \left(\underset{t=1}{\overset{m}{\circledast}} [\![I_t]\!] \right).$$

To prove the theorem, assume first that the resource invariants in I are precise and the $*$ operation is cancellative. In this case we can adapt the proof of soundness of sequential logic with procedures given in Section 5.2.1 to concurrent setting. Let $\mathcal{I} = [\![I]\!]$. We extend the definition of R_i in Section 5.2.1 with predicate transformers $\mathsf{Post}(C)$ for `acquire` and `release` commands defining their thread-local semantics.[6] The definition of the predicate transformer for `acquire(`ℓ_k`)` follows the global ACQUIRE axiom—upon acquiring a lock the thread gets the ownership of the resource invariant associated with the lock:

$$\mathsf{Post}(\mathtt{acquire}(\ell_k), p) = p * \mathcal{I}_k.$$

Consider now the command `release(`ℓ_k`)`. According to the global RELEASE axiom, upon releasing a lock the thread has to give up the ownership of the part of its local state satisfying the lock's resource invariant. When the resource invariant $\mathcal{I}_k$ is precise there may be at most one such part. Thus, when the $*$ operation is cancellative, we can define the transformer as the pointwise lifting of the following function $\mathsf{Post}(\mathtt{release}(\ell_k)) : \Sigma_{\mathbf{e}} \times \Sigma \to D_2$ that removes the part of the state satisfying $\mathcal{I}_k$ and faults if no such part exists:

$$\mathsf{Post}(\mathtt{release}(\ell_k), \xi) = \begin{cases} \{\xi \backslash \mathcal{I}_k\}, & \text{if } (\xi \backslash \mathcal{I}_k)\!\downarrow; \\ \top, & \text{otherwise.} \end{cases}$$

The transformers for `acquire` and `release` defined in this way are local. It is easy to check that Lemmas 5.2 and 5.3 hold with the newly defined R_i.

Note that $\mathsf{Post}(\mathtt{release}(\ell_k))$ is not well-defined when $\mathcal{I}_k$ is imprecise, which makes it impossible to adapt the proof of Section 5.2.1 to concurrent setting in this case. As we show below, the proof of Lemma 5.5 provides an ingredient for a suitable induction hypothesis that can be used when resource invariants may be imprecise to perform the proof directly with respect to the operational semantics, without constructing a denotational one.

We define the functions

$$\nu_k : \Sigma^+_{\mathbf{e}} \times \Sigma \to (\{1, \dots, n\} \rightharpoonup \Sigma^+_{\mathbf{e}}) \times \Sigma,\ k = 1..n;\ \ \nu_0 : \Sigma_{\mathbf{e}} \times \Sigma \to (\{1, \dots, n\} \rightharpoonup \Sigma^+_{\mathbf{e}}) \times \Sigma$$

as follows:

$$\nu_k(\alpha, \sigma) = ([k : \alpha], \sigma),\ k = 1..n;\ \ \nu_0(\eta, \sigma) = ([\,], \sigma)$$

and lift them pointwise to $\mathcal{P}(\Sigma^+_{\mathbf{e}} \times \Sigma)^\top$ and $\mathcal{P}(\Sigma_{\mathbf{e}} \times \Sigma)^\top$, respectively. Let the partial operation $\star$ on $(\{1, \dots, n\} \rightharpoonup \Sigma^+_{\mathbf{e}}) \times \Sigma$ be defined as follows: $(g_1, \sigma_1) \star (g_2, \sigma_2) = (g_1 \uplus g_2, \sigma_1 * \sigma_2)$. We lift $\star$ to $\mathcal{P}((\{1, \dots, n\} \rightharpoonup \Sigma^+_{\mathbf{e}}) \times \Sigma)^\top$ pointwise and use an iterated version of the lifted $\star$:

$$\mathop{\circledast}_{i=1}^{k} = (\{[\,]\} \times e) \star p_1 \star \dots \star p_k.$$

Theorem 5.7 follows from the following lemma.

[6] The thread-local semantics we define here is similar to the local enabling relation in the Brookes's proof of soundness of concurrent separation logic [12].

Lemma 5.8. *If*

$$\{(g_0, \sigma_0)\} \sqsubseteq \left(\bigstar_{t=1}^{n} \nu_t(\mathsf{Lv}_{\mathsf{main}t}) \right) \star \nu_0 \left(\circledast_{t \in \{1,\dots,m\}} \mathcal{I}_t \right),$$

then, whenever $\mathsf{pc}_0, (g_0, \sigma_0) \to_S^* \mathsf{pc}, \zeta$, *we have*

$$\{\!|\zeta|\!\} \sqsubseteq \left(\bigstar_{t=1}^{n} \nu_t(H(\mathsf{pc}(t))) \right) \star \nu_0 \left(\circledast_{t \in \mathsf{Free}(\mathsf{pc})} \mathcal{I}_t \right) \tag{5.31}$$

for $H : N^+ \to \mathcal{P}(\Sigma_{\mathsf{e}}^+ \times \Sigma)^\top$ *defined as follows:*

$$H(v_1 \dots v_k) =$$
$$\bigsqcup \left\{ \left(\bigodot_{t=1}^{k-1} \mathsf{Env}(\{\!|\xi_t|\!\}) \right) \odot \{\!|\xi_k|\!\} \mid \xi_t,\ t = 1..k \textit{ satisfy (5.10) for } \xi_0 \in \mathsf{Lv}_{\mathsf{main}i_1} \right\}.$$

The lemma is a straightforward combination of Lemmas 3.1 and 5.4 (in particular, these lemmas are its special cases). In the statement of the lemma $H(v_1 \dots v_k)$ gives the set of procedure-global, but thread-local states reachable when the corresponding thread is at the call-stack $v_1 \dots v_k$, which are computed as in Lemma 5.4. The set of reachable global states is then computed as in Lemma 3.1 with thread-local states given by H. We omit the proof of the lemma and turn to the more interesting case of imprecise resource invariants.

When the resource invariants I may be imprecise, the proof of Theorem 5.7 is obtained by combining the proofs of soundness given in Sections 5.2.2 and 3.4.3 with the aid of the following lemma, generalising Lemmas 5.5 and 3.1. In this case, the notion of semantic proofs is a combination of procedure-local and thread-local semantic proofs: here we define it as a tuple $(C, G, \mathcal{E}, \mu, \mathcal{I})$ satisfying (5.12)–(5.14) and (3.9)–(3.10) for the corresponding edges in the CFG of C.

Lemma 5.9 (Soundness of the intermediate interpretation). *Consider a semantic procedure context* $\mathcal{E}$ *of the form (5.6) for some set of indices* $\mathcal{R}$. *Assume semantic proofs* $(C_i, G_i^j, \mathcal{E}, \mu(i,j), \mathcal{I})$ *for* $i = 0..l$, $j \in \mathsf{specs}(i)$, *where* $G_i^j(\mathsf{start}_i) \sqsupseteq p_i^j * \mathsf{Lv}_i$ *and* $\mathsf{State}(G_i^j(\mathsf{end}_i)) \sqsubseteq q_i^j$, *and take* $j_1 \in \mathcal{R}$ *such that* $j_1 \in \mathsf{specs}(\mathsf{main}t)$, $t = 1..n$. *If* $g_0 \in \{1, \dots, n\} \to \Sigma_{\mathsf{e}}$ *and* $\sigma_0 \in \Sigma$ *are such that* $g_0(t) \in \mathsf{Lve}_{\mathsf{main}t}$ *for* $t = 1..n$ *and*

$$\{(g_0, \sigma_0)\} \sqsubseteq \left(\bigstar_{t=1}^{n} \nu_t(G_{\mathsf{main}t}^{j_1}(\mathsf{start}_{\mathsf{main}t})) \right) \star \nu_0 \left(\circledast_{t \in \{1,\dots,m\}} \mathcal{I}_t \right),$$

then whenever $\mathsf{pc}_0, (g_0, \sigma_0) \to_S^* \mathsf{pc}, \zeta$, *(5.31) holds for* $H : N^+ \to \mathcal{P}(\Sigma_{\mathsf{e}}^+ \times \Sigma)^\top$ *defined as follows:*

$$H(v_1 \dots v_k) =$$
$$\bigsqcup \left\{ \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot G_{i_k}^{j_k}(v_k) \mid j_t \in \mathsf{specs}(i_t),\ i_t = \mathsf{proc}(v_t),\ t = 1..k \right\}.$$

Proof of Lemma 5.9. As usual, the proof is by induction on the length of the derivation of ζ in the operational semantics of the program S. It largely consists of literally repeating parts of proofs of Lemmas 3.1 and 5.5, therefore, here we present only the case of a sequential command; the other cases are similar. Assume

$$\mathsf{pc}_0, (g_0, \sigma_0) \to_S^* \mathsf{pc}[j : v_1 \dots v_{k-1} v_k], (g, \sigma) \to_S \mathsf{pc}[j : v_1 \dots v_{k-1} v_k], \zeta'$$

and $(v_k, C, v'_k) \in T$, $C \in \mathsf{Seq}$. We need to show that if

$$\{(g, \sigma)\} \sqsubseteq \left(\bigcircledast_{t=1}^{n} \nu_t(H((\mathsf{pc}[j : v_1 \dots v_{k-1} v_k])(t))) \right) \star \nu_0 \left(\circledast_{t \in W} \mathcal{I}_t \right), \tag{5.32}$$

then

$$\{\!|\zeta'|\!\} \sqsubseteq \left(\bigcircledast_{t=1}^{n} \nu_t(H((\mathsf{pc}[j : v_1 \dots v_{k-1} v'_k])(t))) \right) \star \nu_0 \left(\circledast_{t \in W} \mathcal{I}_t \right), \tag{5.33}$$

where $W = \mathsf{Free}(\mathsf{pc}[j : v_1 \dots v_{k-1} v_k]) = \mathsf{Free}(\mathsf{pc}[j : v_1 \dots v_{k-1} v'_k])$. If $H(\mathsf{pc}(t)) = \top$ for some $t \neq j$, or $\mathcal{I}_t = \top$ for some $t \in W$, then (5.33) is trivially true. Otherwise, from (5.32) for some $j_1, \dots, j_k$ such that $j_t \in \mathsf{specs}(i_t)$, $i_t = \mathsf{proc}(v_t)$, $t = 1..k$, we have

$$\{(g(j), \sigma)\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot (G_{i_k}^{j_k}(v_k) * (\mathsf{lift}(\sigma_1)))$$

for $\sigma_1 \in \Sigma$ such that

$$\{(g', \sigma_1)\} \sqsubseteq \left(\bigcircledast_{\substack{1 \le t \le n, \\ t \neq j}} \nu_t(H(\mathsf{pc}(t))) \right) \star \nu_0 \left(\circledast_{t \in W} \mathcal{I}_t \right),$$

where g' is identical to g except it is undefined at j. If $\mu(i_t, j_t, v_t, j_{t+1}) = \top$ for some t, then $H(v_1 \dots v_{k-1} v'_k) = \top$ and (5.33) holds trivially. Otherwise, let $g(j) = \eta_1 \dots \eta_k$, then $\{(\eta_k, \sigma)\} \sqsubseteq G_{i_k}^{j_k}(v'_k) * \mathsf{lift}(\{\sigma_1\} * \{\sigma_2\})$ for some $\sigma_2 \in \Sigma$ such that

$$(\eta_1 \dots \eta_{k-1}, \sigma_2) \in \bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}).$$

Continuing as in case 1 in the proof of Lemma 5.5, we get

$$f_C(\{(\eta_k, \sigma)\}) \sqsubseteq G_{i_k}^{j_k}(v'_k) * \mathsf{lift}(\{\sigma_1\} * \{\sigma_2\}).$$

If $\zeta' = \top$, then $f_C(\{(\eta_k, \sigma)\}) = \top$. From the above it then follows that $G_{i_k}^{j_k}(v'_k) = \top$. Thus, $H(v_1 \dots v_{k-1} v'_k) = \top$, which entails (5.33). If $\zeta' \neq \top$, then $\zeta' = (g'[j : \eta_1 \dots \eta_{k-1} \eta'_k], \sigma')$ and $\{(\eta'_k, \sigma')\} \sqsubseteq f_C(\{(\eta_k, \sigma)\})$. Together with the above, this entails

$$(\eta_1 \dots \eta_{k-1} \eta'_k, \sigma') \in \left(\bigodot_{t=1}^{k-1} \mu(i_t, j_t, v_t, j_{t+1}) \right) \odot (G_{i_k}^{j_k}(v'_k) * \mathsf{lift}(\sigma_1)),$$

which implies (5.33). $\square$

$\mathcal{F}^\sharp(I^0)(G^\sharp, I^\sharp) = (\widetilde{G}^\sharp, \widetilde{I}^\sharp)$, where

- $\widetilde{G}^\sharp(\mathsf{start}_{\mathsf{main}k}) = \mathsf{id}(\mathsf{Lv}^\sharp_{\mathsf{main}k}),\ \ k = 0..n;$
- $\widetilde{G}^\sharp(v', p) = \bigsqcup_{(v,C,v') \in T} g^\sharp_C(G^\sharp(v,p))$, where

$$g^\sharp_C(p) = \begin{cases} f^\sharp_C(p), & \text{if } C \in \mathsf{Seq}; \\ p *^\sharp I^\sharp_k, & \text{if } C \text{ is } \mathtt{acquire}(\ell_k); \\ \mathsf{ThreadLocal}_k(p), & \text{if } C \text{ is } \mathtt{release}(\ell_k); \end{cases}$$

  for every program point $v' \in N$ that is not a start or return point;

- $\widetilde{G}^\sharp(v') = \mathsf{id}\Big(\big\{\mathsf{ProcLocal}_i(q) *^\sharp \mathsf{Lv}^\sharp_i \mid i = \mathsf{calledc}(v) \wedge p \in D^\sharp_2 \cup \{\top\} \wedge$
  $((G^\sharp(v,p) \sqsubset \top \wedge q \in G^\sharp(v,p)) \vee (G^\sharp(v,p) = \top \wedge q = \top))\big\}\Big)$

  for every start point $v' \in N \backslash \{\mathsf{start}_0\}$, where $i = \mathsf{proc}(v')$;

- $\widetilde{G}^\sharp(v', p) = \bigsqcup\{\mathsf{State}^\sharp(G^\sharp(\mathsf{end}_i, \mathsf{ProcLocal}_i(q) *^\sharp \mathsf{Lv}^\sharp_i)) *^\sharp \{\!|\mathsf{Frame}_i(q)|\!\} \mid$
  $(G^\sharp(\mathsf{call}(v'), p) \sqsubset \top \wedge q \in G^\sharp(\mathsf{call}(v'), p)) \vee (G^\sharp(\mathsf{call}(v'), p) = \top \wedge q = \top)\}$

  for every return point $v' \in N$, where $i = \mathsf{calledr}(v')$.

- $\widetilde{I}^\sharp_k = I^0_k \sqcup \bigsqcup\{\mathsf{Protected}_k(G^\sharp(v,q)) \mid q \in D^\sharp \cup \{\top\} \wedge (v, \mathtt{release}(\ell_k), v') \in T\}$

  for every lock ℓ_k.

Figure 5.5: Interprocedural thread-modular analysis

5.4.2 Analysis

We assume the setting of Section 5.3. The interprocedural thread-modular analysis is obtained by composing the analyses defined in Sections 5.3 and 3.3. The analysis operates on the domain

$$\widehat{D}^\sharp = (N \to ((D^\sharp_2 \cup \{\top\}) \to \mathcal{P}(D^\sharp_2)^\top)) \times (\mathcal{P}(D^\sharp_2)^\top)^m.$$

Given a vector $I^0 \in (\mathcal{P}(D^\sharp_2)^\top)^m$ of initial approximations of resource invariants, it is defined by the functional $\mathcal{F}^\sharp(I^0) : \widehat{D}^\sharp \to \widehat{D}^\sharp$ in Figure 5.5. The analysis is a straightforward combination of the analyses in Figures 5.3 and 3.3. In addition to the ingredients of the local RHS analysis of Section 5.3, we parameterise it with functions $\mathsf{ThreadLocal}_k : D^\sharp_2 \to D^\sharp_2$ and $\mathsf{Protected}_k : D^\sharp_2 \to D^\sharp_2$ for $k = 1..m$ defining the splitting of the state at `release` commands. We require that they satisfy an analogue of (3.3) for the domain D_2 and that for any $p \in D_2$ and $k = 1..m$, $\gamma(\mathsf{Protected}_k(p))$ have an empty environment. We lift $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ to $\mathcal{P}(D^\sharp_2)^\top$ pointwise.

Using Lemma 5.9, the soundness of the analysis is justified by combining the proofs in Sections 5.3 and 3.4.2. As before, the soundness statement is a combination of the corresponding theorems for concurrent programs with static locks (Theorem 3.2) and for sequential programs with procedures (Theorem 5.6).

Theorem 5.10 (Soundness of the analysis). *Let $(G^\sharp, I^\sharp)$ be a fixed point of $\mathcal{F}^\sharp(I^0)$. If*

$$\{(g_0, \sigma_0)\} \sqsubseteq \left(\mathop{\circledast}_{t=1}^{n} \nu_t(\mathsf{Lv}_{\mathsf{main}t}) \right) \star \nu_0 \left(\mathop{\circledast}_{t \in \{1,\dots,m\}} \gamma(I_t^0) \right),$$

then, whenever $\mathsf{pc}_0, (g_0, \sigma_0) \to_S^* \mathsf{pc}, \zeta$, we have

$$\{\!|\zeta|\!\} \sqsubseteq \left(\mathop{\circledast}_{t=1}^{n} \nu_t(H(\mathsf{pc}(t))) \right) \star \nu_0 \left(\mathop{\circledast}_{t \in \mathsf{Free}(\mathsf{pc})} \gamma(I_t^\sharp) \right)$$

for $H : N^+ \to \mathcal{P}(\Sigma_{\mathbf{e}}^+ \times \Sigma)^\top$ defined as follows:

$$H(v_1 \dots v_k) = \bigsqcup \left\{ \left(\bigodot_{t=1}^{k-1} \gamma(\mathsf{Frame}_{i_{t+1}}(w_t)) \right) \odot \gamma(w_k) \mid w_1, \dots, w_k \in D_2^\sharp \cup \{\top\} \text{ satisfy (5.28) for } \mathsf{pre} = e^\sharp \right\}.$$

5.5 Logical variables and cutpoints

In this section we prove the soundness of the proof rules for manipulating logical variables, Exists and Forall, for programs with procedures and show how the interprocedural analysis of Section 5.3 can handle cutpoints precisely using logical variables.

In the setting of Section 5.1, assume that Σ is an algebra with logical variables (Section 2.1.3), i.e., $\Sigma = \Sigma' \times \text{Ints}$, then so is $\Sigma_{\mathbf{e}} \times \Sigma = (\Sigma_{\mathbf{e}} \times \Sigma') \times \text{Ints}$. We also assume that $\mathcal{P}(\Sigma')$ has a unit element e' such that $e' \times \text{Ints} = e$. We require that the functions f_C for $C \in \mathsf{Seq}$ are lifted from functions on $\Sigma_{\mathbf{e}} \times \Sigma'$.

5.5.1 Logic

Consider the logic for sequential programs with procedures of Section 5.2. Under the above conditions, the proof of Theorem 5.1 given in Section 5.2.1 can be easily extended to the case when the logic contains the rules Exists and Forall: the functions g_i can be represented as lifted from local functions on Σ', which allows us to establish Lemma 5.2 for the extended logic.

We now extend the proof of soundness given in Section 5.2.2 to the case when the logic contains the rule Exists (as is the case with Conj, the proof cannot be extended to prove the soundness of Forall). Later in this section, we use the new proof to establish the soundness of Exists in the concurrent setting for the case when resource invariants may be imprecise.

Our goal is to reuse Lemma 5.5 and the notion of semantic proofs introduced in Section 5.2.2. To prove the soundness of Exists, we would like to establish an analogue of Lemma 3.11(i) that constructs a semantic proof for the conclusion of the rule out of a semantic proof for its premiss. Unfortunately, the straightforward analogue of Lemma 3.11(i) does not hold for semantic proofs considered here. Indeed consider the derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \{P\}\ \mathtt{f}\ \{Q\}}\ \textsc{ProcCall}}{\Gamma \vdash \{P * R\}\ \mathtt{f}\ \{Q * R\}}\ \textsc{Frame}}{\Gamma \vdash \{\exists X.\, P * R\}\ \mathtt{f}\ \{\exists X.\, Q * R\}}\ \textsc{Exists}$$

where $\Gamma = \big\{\{P\}\ \mathtt{f}\ \{Q\}\big\}$.

We cannot in general represent $\exists X.\, P * R$ in the form $P * R'$ and, hence, cannot construct a semantic proof for the conclusion of Exists in this derivation over the semantic procedure context corresponding to Γ out of a semantic proof for the premiss. The solution is to use the semantic counterpart of the equivalence

$$\exists X.\, P * R \Leftrightarrow \bigvee_{u \in \text{Values}} (P[u/X] * R[u/X]),$$

which allows us to construct a semantic proof over an extended semantic procedure context that includes all variants of the summaries from Γ where values are substituted for logical variables in all possible ways. Furthermore, we show below that out of such semantic proofs we can construct semantic proofs justifying the specifications in the extended procedure context. This set of semantic proofs can then be used in Lemma 5.5. Note that, strictly speaking, such a proof is not analogous to a proof of the admissibility of Exists, since we are effectively converting derivations for the whole program, not just for the part of the derivation dealing with the command Exists is applied to. This is because in the case of imprecise resource invariants we do not have a thread-local semantics of the procedure call command validating Exists akin to the one we used in Section 5.2.1.

Before presenting the proof, we first give some auxiliary definitions. Let PartInts $=$ LVars $\rightharpoonup$ Values be the set of partial interpretations, giving values to some of logical variables. For $\mathbf{i} \in$ Ints and $\tau \in$ PartInts, we denote with $\mathbf{i}[\tau]$ the interpretation $\mathbf{i}[\tau]$ such that for any $X \in$ LVars

$$(\mathbf{i}[\tau])(X) = \begin{cases} \tau(X), & \text{if } \tau(X){\downarrow}\,; \\ \mathbf{i}(X), & \text{if } \tau(X){\uparrow}\,. \end{cases}$$

We define the operation $\oplus$: PartInts $\times$ PartInts $\to$ PartInts as follows: for any $X \in$ LVars

$$(\tau_1 \oplus \tau_2)(X) = \begin{cases} \tau_1(X), & \text{if } \tau_1(X){\downarrow}\,; \\ \tau_2(X), & \text{if } \tau_1(X){\uparrow} \text{ and } \tau_2(X){\downarrow}\,; \\ \text{undefined}, & \text{if } \tau_1(X){\uparrow} \text{ and } \tau_2(X){\uparrow}\,. \end{cases}$$

The operation takes the union of two partial interpretations giving priority to the first argument in cases of overlap. For $p \in D_2$, $X \in \text{LVars}$, and $\tau \in \text{PartInts}$ let

$$\mathsf{Exists}(X, p) = \begin{cases} \{(\xi', \mathbf{i}) \mid \exists u \in \text{Values}.\, (\xi', \mathbf{i}[X : u]) \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top \end{cases} \tag{5.34}$$

and

$$\mathsf{Subst}(\tau, p) = \begin{cases} \{(\xi', \mathbf{i}) \mid (\xi', \mathbf{i}[\tau]) \in p\}, & \text{if } p \neq \top; \\ \top, & \text{if } p = \top \end{cases} \tag{5.35}$$

be the semantic counterparts of existentially quantifying a logical variable and substituting logical variables for their values, respectively. We extend Exists to sets of variables in the expected way and define Exists and Subst on $\mathcal{P}(\Sigma_{\mathbf{e}}^{+} \times \Sigma)^{\top}$ analogously. For the future, we note the following properties of Exists and Subst:

$$\forall \tau_1, \tau_2 \in \text{PartInts}.\, \forall p \in D_2.\, \mathsf{Subst}(\tau_1, \mathsf{Subst}(\tau_2, p)) = \mathsf{Subst}(\tau_2 \oplus \tau_1, p); \tag{5.36}$$

$$\forall X \in \text{LVars}.\, \forall p \in D_2.\, \mathsf{Exists}(X, p) = \bigsqcup_{u \in \text{Values}} \mathsf{Subst}([X : u], p). \tag{5.37}$$

Consider a semantic procedure context $\mathcal{E}$ of the form (5.6) for some $\mathcal{R}$. We construct a new context $\mathcal{E}'$ out of it, consisting of summaries in $\mathcal{E}$ with logical variables in pre- and postconditions substituted for values in all possible ways. Formally, let $\mathcal{R}' = \mathcal{R} \times \text{PartInts}$ and $\mathsf{specs}'(i) = \mathsf{specs}(i) \times \text{PartInts}$ for $i = 0..l$. We define

$$p_i^{(j,\tau)} = \mathsf{Subst}(\tau, p_i^j),\ q_i^{(j,\tau)} = \mathsf{Subst}(\tau, q_i^j),\ i = 0..l,\ (j, \tau) \in \mathsf{specs}'(i)$$

and

$$\mathcal{E}' = \big\{\{p_i^{(j,\tau)}\}\ \mathtt{f}_i\ \{q_i^{(j,\tau)}\} \mid i = 0..l,\ (j, \tau) \in \mathsf{specs}'(i)\big\}. \tag{5.38}$$

The proof of soundness of the logic with the rule Exists relies on the following lemma establishing that:

(i) semantic proofs over the semantic procedure context $\mathcal{E}$ can be recast as semantic proofs over the larger context $\mathcal{E}'$;

(ii) semantic proofs over $\mathcal{E}'$ are closed under substituting values for logical variables in their semantic annotations;

(iii) semantic proofs over $\mathcal{E}'$ are closed under existentially qualifying logical variables in their semantic annotations.

The latter statement is an analogue of Lemma 3.11(i).

Lemma 5.11. *(i) If $(C, G, \mathcal{E}, \mu)$ is a semantic proof, then so is $(C, G, \mathcal{E}', \mu')$, where*

$$\forall v, j, \tau''.\, \mu'(v, (j, \tau'')) = \begin{cases} \mu(v, j), & \text{if } \tau'' = [\,]; \\ \bot, & \text{otherwise.} \end{cases}$$

(ii) For any $\tau \in \text{PartInts}$, *if* $(C, G, \mathcal{E}', \mu)$ *is a semantic proof, then so is* $(C, G', \mathcal{E}', \mu')$, *where* $\forall v.\, G'(v) = \mathsf{Subst}(\tau, G(v))$ *and*

$$\forall v, j, \tau''.\, \mu'(v, (j, \tau'')) = \bigsqcup \{\mathsf{Subst}(\tau, \mu(v, (j, \tau'))) \mid \tau' \oplus \tau = \tau''\}.$$

(iii) For any $X \in \text{LVars}$, *if* $(C, G, \mathcal{E}', \mu)$ *is a semantic proof, then so is* $(C, G', \mathcal{E}', \mu')$, *where* $\forall v.\, G'(v) = \mathsf{Exists}(X, G(v))$ *and*

$$\forall v, j, \tau''.\, \mu'(v, (j, \tau'')) = \bigsqcup \{\mathsf{Subst}([X : u], \mu(v, (j, \tau'))) \mid \tau' \oplus [X : u] = \tau''\}.$$

Proof. Consider an edge (v, C', v') in the CFG of the command C. When $C' \in \mathsf{Seq}$, in all cases (5.12) for the new semantic proof follows from the fact that f_C are lifted from functions on $\Sigma_{\mathbf{e}} \times \Sigma'$. Consider the case when C' is $\mathtt{f}_i$. We show that (5.13) holds for the new semantic proof in cases (ii) and (iii) (case (i) is easy and (5.14) is established similarly).

(ii) Using the definition of G' and μ' and (5.36), we get:

$$\begin{aligned}
G'(v) &= \mathsf{Subst}(\tau, G(v)) \\
&\sqsubseteq \mathsf{Subst}\left(\tau, \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} p_i^{(j,\tau')} * \mu(v', (j, \tau'))\right) \\
&= \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} \mathsf{Subst}(\tau, p_i^{(j,\tau')}) * \mathsf{Subst}(\tau, \mu(v', (j, \tau'))) \\
&= \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} p_i^{(j,\tau' \oplus \tau)} * \mathsf{Subst}(\tau, \mu(v', (j, \tau'))) \\
&= \bigsqcup_{(j,\tau'') \in \mathsf{specs}'(i)} p_i^{(j,\tau'')} * \mu'(v', (j, \tau''))
\end{aligned}$$

(iii) In this case, using (5.37) we similarly obtain:

$$\begin{aligned}
G'(v) &= \mathsf{Exists}(X, G(v)) \\
&\sqsubseteq \mathsf{Exists}\left(X, \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} p_i^{(j,\tau')} * \mu(v', (j, \tau'))\right) \\
&= \bigsqcup_{u \in \text{Values}} \mathsf{Subst}\left([X : u], \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} p_i^{(j,\tau')} * \mu(v', (j, \tau')\right) \\
&= \bigsqcup_{u \in \text{Values}} \bigsqcup_{(j,\tau') \in \mathsf{specs}'(i)} p_i^{(j,\tau' \oplus [X:u])} * \mathsf{Subst}([X : u], \mu(v', (j, \tau'))) \\
&= \bigsqcup_{(j,\tau'') \in \mathsf{specs}'(i)} p_i^{(j,\tau'')} * \mu'(v', (j, \tau'')))
\end{aligned}$$

□

We can now modify the proof of Theorem 5.1 given in Section 5.2.2 to include the rule Exists. Consider the proof (5.5) of the program S. As before, let $\mathcal{E}$ be the semantic procedure context constructed out of the procedure context Γ represented in the form (5.6) with $\mathcal{R} = \mathbb{N}$ and let $\mathcal{E}'$ be defined by (5.38) with $\mathcal{R}' = \mathcal{R} \times \text{PartInts}$. Using Lemma 5.11(i,iii), we can show that from a derivation of a triple $\Gamma \vdash \{P * \mathsf{Vars}(\vec{\mathtt{x}}_i)\}\ C_i\ \{Q * \mathbf{true_e}\}$ that does not use Conj but possibly uses Exists, we can construct a semantic proof $(C_i, G, \mathcal{E}', \mu)$ for some G and μ such that $G(\mathsf{start}_i) = [\![P]\!] * \mathsf{Lv}_i$, $\mathsf{State}(G(\mathsf{end}_i)) \sqsubseteq [\![Q]\!]$, $\forall v.\, G(v) \sqsubset \top$, and $\forall v, j.\, \mu(v, j) \sqsubset \top$. By Lemma 5.11(ii), for any $\tau \in \text{PartInts}$ from this semantic proof we can construct a semantic proof $(C_i, G', \mathcal{E}', \mu')$ for some μ', where $\forall v.\, G'(v) = \mathsf{Subst}(\tau, G(v))$ and hence, $G'(\mathsf{start}_i) = \mathsf{Subst}(\tau, [\![P]\!]) * \mathsf{Lv}_i$ and $\mathsf{State}(G'(\mathsf{end}_i)) \sqsubseteq \mathsf{Subst}(\tau, [\![Q]\!])$. The set of semantic proofs constructed in this way from (5.5) satisfies the conditions of Lemma 5.5, which implies Theorem 5.1.

Concurrent setting. Consider the logic for concurrent programs with procedures (Section 5.4.1), where the denotations of resource invariants do not depend on logical variables. In the case where resource invariants are required to be precise and the $*$ operation cancellative, the proof of Theorem 5.7 given in Section 5.4.1 is adapted to the logic with the rules Exists and Forall in the same way as the proof of Theorem 5.1 in Section 5.2.1 is adapted earlier in this section. In the case where resource invariants may be imprecise, we can establish Theorem 5.7 for the logic with the rule Exists in the same way as Theorem 5.1 earlier in this section. This relies on the fact that Lemma 5.11 holds for semantic proofs introduced in Section 5.4.1.

5.5.2 Analysis

We now modify the local RHS analysis to give precise treatment to cutpoints.

Analysis formulation. Consider the local RHS analysis of Section 5.3. We additionally parameterise it with two auxiliary functions:

- $\mathsf{Cutpoints}_i : D_2^\sharp \to \mathcal{P}(\text{LVars})$, $i = 0..l$ such that $\mathsf{Cutpoints}_i(p)$ returns the set of logical variables to represent cutpoints arising at a call point of procedure $\mathtt{f}_i$ when the state at the call point is p. Let $\mathsf{Cutpoints}(\top) = \emptyset$.
- $\mathsf{Exists}^\sharp : \mathcal{P}(\text{LVars}) \times D_2^\sharp \to D_2^\sharp$ such that $\mathsf{Exists}^\sharp(\vec{X}, p)$ over-approximates the meaning of existentially qualifying the variables $\vec{X}$ in p:

  $$\forall p \in D_2^\sharp.\, \mathsf{Exists}(\vec{X}, \gamma(p)) \sqsubseteq \gamma(\mathsf{Exists}^\sharp(\vec{X}, p)).$$

  We lift $\mathsf{Exists}^\sharp$ to $\mathcal{P}(D_2^\sharp)^\top$ pointwise.

We then replace condition (5.24) with

$$\forall p \in D_2^\sharp.\, \gamma(p) \sqsubseteq \mathsf{Exists}(\mathsf{Cutpoints}_i(p), \gamma(\mathsf{ProcLocal}_i(p)) * \gamma(\mathsf{Frame}_i(p))). \qquad (5.39)$$

The new analysis is defined by the functional in Figure 5.3 with the equation for return points $v' \in N$ replaced by the following one:

$$\begin{aligned}\widetilde{G}^\sharp(v',p) = \bigsqcup\{&\mathsf{Exists}^\sharp(\mathsf{Cutpoints}_i(q), \mathsf{State}^\sharp(G^\sharp(\mathsf{end}_i, \mathsf{ProcLocal}_i(q) *^\sharp \mathsf{Lv}_i^\sharp)) *^\sharp \{\!\!|\mathsf{Frame}_i(q)|\!\!\})\mid\\ &(G^\sharp(\mathsf{call}(v'),p) \sqsubset \top \wedge q \in G^\sharp(\mathsf{call}(v'),p)) \vee (G^\sharp(\mathsf{call}(v'),p) = \top \wedge q = \top)\},\end{aligned} \tag{5.40}$$

where $i = \mathsf{calledr}(v')$. Intuitively, the analysis allows $\mathsf{ProcLocal}_i$ and Frame_i to introduce logical variables denoting cutpoints that allow us to maintain pointers between the frame and the local heap. After conjoining the postcondition of the procedure to the frame, these variables are eliminated using $\mathsf{Exists}^\sharp$.

Determining heap splittings. To define $\mathsf{ProcLocal}_k$ and Frame_k, we can use the same heuristic as in Section 5.3, modified to account for the treatment of cutpoints. We illustrate its implementation using the the example of the domain SLL. Again, consider a variation of SLL where the assertions are interpreted with respect to the domain RAM_2 of Section 5.1. We define $\mathsf{ProcLocal}_k$ and Frame_k for this domain by modifying equations (5.26) and (5.27) to treat cutpoints precisely:

$$\begin{aligned}&\mathsf{ProcLocal}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) =\\ &\qquad\qquad \mathsf{can}(\mathsf{Globals}(O) \Vdash \exists \vec{X}.\, ((\vec{Z} = \vec{\mathsf{z}}) \wedge \mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Globals}(O)))[\vec{Y}/\vec{\mathsf{y}}]),\end{aligned}$$

$$\begin{aligned}&\mathsf{Frame}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) =\\ &\qquad\qquad \mathsf{can}(\mathsf{Locals}(O) \Vdash \exists \vec{X}.\, ((\vec{Y} = \vec{\mathsf{y}}) \wedge \mathsf{P} \wedge \mathsf{Unreach}(\mathsf{S}, \mathsf{Globals}(O)))[\vec{Z}/\vec{\mathsf{z}}]),\end{aligned}$$

and $\mathsf{Cutpoints}_k(O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S}) = \vec{Y} \cup \vec{Z}$ for fresh $\vec{Y}, \vec{Z}$, where

$$\vec{\mathsf{y}} = \mathsf{vars}(\mathsf{P} \wedge \mathsf{Reach}(\mathsf{S}, \mathsf{Globals}(O))) \cap \mathsf{Locals}(O)$$

and

$$\vec{\mathsf{z}} = \mathsf{vars}(\mathsf{P} \wedge \mathsf{Unreach}(\mathsf{S}, \mathsf{Globals}(O))) \cap \mathsf{Globals}(O).$$

Here $\vec{Y}$ record the values of cutpoints in the local heap; the frame asserts that they are equal to the corresponding local variables of the caller. We also choose to track how the caller changes the values of the global variables mentioned in the frame, thus, $\vec{Z}$ record their values before the procedure called starts executing. It is easy to check that $\mathsf{ProcLocal}_k$, Frame_k, and $\mathsf{Cutpoints}_k$ defined in this way satisfy (5.39). We also define $\mathsf{Exists}^\sharp$ as the existential qualifications of symbolic heaps:

$$\mathsf{Exists}^\sharp(\vec{Y}, (O \Vdash \exists \vec{X}.\, \mathsf{P} \wedge \mathsf{S})) = \mathsf{can}(O \Vdash \exists \vec{X}, \vec{Y}.\, \mathsf{P} \wedge \mathsf{S}).$$

Again, consider the symbolic heap

$$P = (\mathtt{x}, \mathtt{y}, \mathtt{u}, \mathtt{v} \Vdash \mathsf{ls}(\mathtt{x}, \mathtt{u}) * \mathsf{ls}(\mathtt{u}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL}) * \mathsf{ls}(\mathtt{v}, \mathtt{x})),$$

where $\mathtt{x}$ and $\mathtt{y}$ are global, and $\mathtt{u}$ and $\mathtt{v}$ are local. In this case

$$\mathsf{ProcLocal}_k(P) = (\mathtt{x}, \mathtt{y} \Vdash \mathtt{x} = Z \wedge \mathsf{ls}(\mathtt{x}, Y) * \mathsf{ls}(Y, \mathtt{NULL}) * \mathsf{ls}(\mathtt{y}, \mathtt{NULL})),$$

$\mathsf{Frame}_k(P) = (\mathtt{u}, \mathtt{v} \Vdash \mathtt{u} = Y \wedge \mathsf{ls}(\mathtt{v}, Z))$, and $\mathsf{Cutpoints}_k(P) = \{Y, Z\}$.

Note that for simplicity of presentation the above definition precisely handles only the cutpoints that arise from tracking values of global variables and from stack sharing, i.e., in the situation when a location in the local heap is equal to the value of a local variable of the caller. The other kind of cutpoints result from heap sharing, i.e., in the situation when a location in the local heap is equal to the contents of a location in the frame and distinct from the global variables and the local variables of the caller. Such cutpoints are defined by quantified variables in the symbolic heap at the call-site, e.g., Z in the heap $\mathtt{x}, \mathtt{u} \Vdash \exists Z.\, \mathsf{ls}(\mathtt{x}, Z) * \mathsf{ls}(\mathtt{u}, Z)$, where $\mathtt{x}$ is a global variable and $\mathtt{u}$ is a local variable of the caller. This kind of cutpoints can be handled precisely in a similar fashion, i.e., by replacing them with free logical variables. In the example, the local heap is $\mathtt{x} \Vdash \mathsf{ls}(\mathtt{x}, Z)$ and the frame $\mathtt{u} \Vdash \mathsf{ls}(\mathtt{u}, Z)$.

To prevent the number of free logical variables introduced by the analysis from growing unboundedly, we can easily combine the implementations of $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$ given here and in Section 5.3 into one that abstracts all the cutpoints away (by computing the splitting as in Section 5.3) if their number at the call point exceeds a predefined bound. We observe that the pathological cases where the number of cutpoints grows unboundedly while analysing recursive procedures is rarely encountered in practice (especially if dead variable analysis is used to prevent our analysis from tracking the values of dead local variables). Even when the number of cutpoints in a symbolic heap at the call point exceeds the predetermined bound, our analysis is still able to obtain some information (though not the most precise).

We have implemented an instantiation of the analysis presented here with the domain SLL. Experimental results on list-manipulating programs are reported in [31].

Soundness. We derive the soundness statement for the analysis with cutpoints by modifying the proof of soundness for the analysis in Section 5.3 using the same ideas as in the proof of soundness of the rule EXISTS given in Section 5.5.1.

As before, we show that the results of the analysis generate an instance of the procedure-local interpretation introduced in Section 5.2.2. Let $G^\sharp \in \widehat{D}^\sharp$ be a fixed point of $\mathcal{F}^\sharp_{\mathsf{pre}}$. We define $\mathcal{R}$, specs, p^r_i, q^r_i, $\mathcal{E}$, and G^r_i as in Section 5.3, and $\mathcal{R}'$, specs', and $\mathcal{E}'$ as in Section 5.5.1. For $i = 0..l$, $r \in \mathsf{specs}(i)$ let $\mu'(i, r) : N_i \times \mathcal{R}' \to D_2$ be defined as follows: if v is a return point, then

$$\mu'(i, r, v, (r', \tau')) = \bigsqcup \big\{\mathsf{Subst}(\tau', \gamma(\mathsf{Frame}_t(w))) \mid \{\!\{w\}\!\} \sqsubseteq G^\sharp(\mathsf{call}(v), r *^\sharp \mathsf{Lv}^\sharp_i) \wedge \\ r' = \mathsf{ProcLocal}_t(w) \wedge \mathsf{Cutpoints}_t(w) = \mathsf{dom}(\tau')\big\},$$

where $t = \mathsf{calledr}(v)$; otherwise, $\mu(i, r, v, (r', \tau'))$ is assigned an arbitrary value. As before, it is easy to show that $(C_i, G_i^r, \mathcal{E}', \mu'(i, r))$, $i = 0..l$, $r \in \mathsf{specs}(i)$ are semantic proofs such that $G_i^r(\mathsf{start}_i) \sqsupseteq p_i^r * \mathsf{Lv}_i$ and $\mathsf{State}(G_i^r(\mathsf{end}_i)) \sqsubseteq q_i^r$. By Lemma 5.11(ii) we have that $(C_i, G_i^{(r,\tau)}, \mathcal{E}', \mu''(i, (r, \tau)))$, $i = 0..l$, $r \in \mathsf{specs}(i)$ are semantic proofs, where for all v we have $G_i^{(r,\tau)}(v) = \mathsf{Subst}(\tau, G_i^r(v))$ and

$$\begin{aligned}\mu''(i, (r, \tau), v, (r', \tau')) &= \bigsqcup\{\mathsf{Subst}(\tau, \mu'(i, r, v, (r', \tau''))) \mid \tau'' \oplus \tau = \tau'\} \\ &= \bigsqcup\{\mathsf{Subst}(\tau', \gamma(\mathsf{Frame}_t(w))) \mid \{\!\!\{w\}\!\!\} \sqsubseteq G^\sharp(\mathsf{call}(v), r *^\sharp \mathsf{Lv}_i^\sharp) \wedge \\ &\qquad r' = \mathsf{ProcLocal}_t(w) \wedge \tau'' \oplus \tau = \tau' \wedge \mathsf{Cutpoints}_t(w) = \mathsf{dom}(\tau'')\},\end{aligned}$$

for $t = \mathsf{calledr}(v)$. In particular, $G_i^{(r,\tau)}(\mathsf{start}_i) \sqsupseteq p_i^{(r,\tau)} * \mathsf{Lv}_i$ and $\mathsf{State}(G_i^{(r,\tau)}(\mathsf{end}_i)) \sqsubseteq q_i^{(r,\tau)}$. Thus, the semantic proofs constructed in this way satisfy the conditions of Lemma 5.5.

Now assume $\{(\eta_0, \sigma_0)\} \sqsubseteq \gamma(\mathsf{pre}) * \mathsf{Lv}_0$, and $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \dots v_k, \zeta$. By Lemma 5.5, for some $r_1, \dots, r_k$ and $\tau_1, \dots, \tau_k$ such that $r_1 = \mathsf{pre}$, $\tau_1 = [\,]$, $(r_t, \tau_t) \in \mathsf{specs}'(i_t)$, $i_t = \mathsf{proc}(v_t)$, $t = 1..k$, we have

$$\{\!\!\{\zeta\}\!\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mu''(i_t, (r_t, \tau_t), v_t, (r_{t+1}, \tau_{t+1}))\right) \odot \mathsf{Subst}(\tau_k, \gamma(G^\sharp(v_k, r_k *^\sharp \mathsf{Lv}_{i_k}^\sharp))).$$

According to the definition of μ'', this is the case if and only if for some $w_1, \dots, w_k \in D^\sharp \cup \{\top\}$ satisfying (5.28) and $\tau_2', \dots, \tau_k' \in$ PartInts such that $\tau_{t+1}' \oplus \tau_t = \tau_{t+1}$, $t = 1..(k-1)$ and $\mathsf{Cutpoints}_{i_{t+1}}(w_t) = \mathsf{dom}(\tau_{t+1}')$, $t = 1..(k-1)$ we have

$$\{\!\!\{\zeta\}\!\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mathsf{Subst}(\tau_{t+1}, \gamma(\mathsf{Frame}_{i_{t+1}}(w_t)))\right) \odot \mathsf{Subst}(\tau_k, \gamma(w_k)).$$

Let $\tau_1' = [\,]$, then $\tau_t = \bigoplus_{j=0}^{t-1} \tau_{t-j}'$, $t = 1..k$. Then we can transform the above inequality to

$$\{\!\!\{\zeta\}\!\!\} \sqsubseteq \left(\bigodot_{t=1}^{k-1} \mathsf{Subst}\left(\bigoplus_{j=0}^{t} \tau_{t-j+1}', \gamma(\mathsf{Frame}_{i_{t+1}}(w_t))\right)\right) \odot \mathsf{Subst}\left(\bigoplus_{j=0}^{k-1} \tau_{k-j}', \gamma(w_k)\right).$$

Let us define the sequence u_t, $t = 1..k$ as follows:

$$u_k = \gamma(w_k); \quad u_{t-1} = \mathsf{Subst}(\tau_t', \gamma(\mathsf{Frame}_{i_t}(w_t)) \odot u_t), \; t = 2..k.$$

Then the above inequality is equivalent to $\{\!\!\{\zeta\}\!\!\} \sqsubseteq u_1$. This holds for some $\tau_2', \dots, \tau_k' \in$ PartInts such that $\tau_{t+1}' \oplus \tau_t = \tau_{t+1}$, $t = 1..(k-1)$ and $\mathsf{Cutpoints}_{i_{t+1}}(w_t) = \mathsf{dom}(\tau_{t+1}')$, $t = 1..(k-1)$ if and only if $\{\!\!\{\zeta\}\!\!\} \sqsubseteq z_1$, where z_1 is defined using the sequence z_t, $t = 1..k$:

$$z_k = \gamma(w_k); \quad z_{t-1} = \mathsf{Exists}(\mathsf{Cutpoints}_{i_t}(w_{t-1}), \gamma(\mathsf{Frame}_{i_t}(w_t)) \odot u_t), \; t = 2..k. \tag{5.41}$$

We have thus proved the following.

Theorem 5.12 (Soundness of the analysis). *Let $G^\sharp \in \widehat{D}^\sharp$ be a fixed point of $\mathcal{F}^\sharp(\mathsf{pre})$. If $(\eta_0, \sigma_0) \in \Sigma_{\mathbf{e}} \times \Sigma$ is such that $\{(\eta_0, \sigma_0)\} \sqsubseteq \gamma(\mathsf{pre}) * \mathsf{Lv}_0$, then whenever $\mathsf{start}_0, (\eta_0, \sigma_0) \to_S^* v_1 \ldots v_k, \zeta$, for some $w_1, \ldots, w_k \in D_2^\sharp \cup \{\top\}$ satisfying (5.28) we have $\{\!|\zeta|\!\} \sqsubseteq z_1$, where z_1 is defined by (5.41).*

The soundness statement is formulated as in Theorem 5.6, except now we existentially quantify the logical variables denoting cutpoints at the points of their introduction.

Concurrent setting. The interprocedural thread-modular analysis with precise treatment of cutpoints is defined by modifying the analysis of Section 5.4.2, replacing the equation for return nodes in Figure 5.5 with (5.40) (we assume the ingredients introduced in this section). The soundness of the analysis is formulated similarly to Theorem 5.10.

5.6 Related work

The results in this chapter build on several pieces of prior work: the proof rules for procedures in Hoare logic [42], Reps-Horwitz-Sagiv algorithm [64], and the idea of localisation in interprocedural analysis [67]. Our contributions are:

- to generalise these results and their prior usage in heap analysis [67, 70, 69, 31, 81, 82] into a uniform framework that can be instantiated with different abstract domains and localisation schemes;
- to develop modular reasoning methods for procedures in concurrent programs, including the case when resource invariants may be imprecise.

As shown in [31], the instantiation of our framework with the domain SLL that we presented in Section 5.3 yields a local interprocedural heap analysis that is more efficient than earlier analyses based on TVLA [70, 69]. Splitting the heap at a procedure call is delicate with the TVLA reachability-based representation as it significantly alters the reachability information. A consequence of this is that accurate treatment of cutpoints is expensive (e.g., [70] considers only programs without cutpoints, and [69] abstracts all cutpoints into a single cutpoint). In abstract domains based on separation logic splitting the heap is easy and (bounded numbers of) cutpoints can be represented by logical variables. An instantiation of our analysis with the domain CDS has been used and optimised by Yang et al. [81, 82], who report experimental results on programs of up to 10k lines of code.

Rinetzky et al. [67] were the first to use localisation in an interprocedural heap analysis, passing the part of the heap reachable from the actual parameters to the callee. The proof of soundness of their analysis handles only this particular localisation scheme defined in the concrete domain (in our terminology, $\mathsf{ProcLocal}_k$ and Frame_k have to satisfy analogues of (3.6) and (3.7)). Our framework allows any localisation schemes defined in the abstract domain, thus allowing flexibility in the implementation of $\mathsf{ProcLocal}_k$ and Frame_k (in particular, they do not have to compute reachability precisely; cf. the discussion at the end of Section 3.3.2).

As we mentioned at the beginning of this chapter, the localisation scheme based on reachability from actual parameters and global variables is not optimal: sometimes the local heap it computes contains more memory cells than what the procedure actually accesses (i.e., more than its footprint), which leads to analysing the procedure more times than necessary. Calcagno et al. [13] have recently proposed a more local interprocedural analysis that discovers approximations to footprints of every procedure in a program and employs a variant of the local RHS analysis.

There exist other approaches to interprocedural heap analysis that are not based on the version of the RHS algorithm we use. For example, Rinetzky and Sagiv [68] explicitly track the shape of the call-stack, and Guo et al. [37] modify the RHS algorithm to use inductive recursion synthesis for constructing summaries.

To keep presentation simple, in this chapter we considered only parameterless procedures. Proof rules for procedures with parameters in the case when variables are treated as resource were proposed by Parkinson et al. [62]. Similar rules could be derived from our proof rules for parameterless procedures and the desugaring of parameter passing we use (Section 5.1).

Chapter 6

Threads

So far, concurrent programs in this dissertation consisted of a top-level parallel composition of several threads. In this chapter we consider dynamic thread creation, adding a command for forking a new thread to the programming language with storable locks of Chapter 4. We present a logic for modular reasoning about programs in this language and a corresponding analysis, extending the logic and the analysis developed in Chapter 4. As before, we have two variants of the logic—with and without the conjunction rule, with the former variant placing restrictions on preconditions of threads.

Constructing the logic and the analysis for dynamic thread creation and proving them sound uses the methods proposed in the previous chapters. Our goal here is to, first, develop aids for reasoning about a more powerful programming construct than parallel composition, and second, to show that the methods developed in this dissertation can be applied to constructs other than the ones they were originally proposed for.

6.1 Programming language and semantics

Programming language. We extend the programming language of Section 2.1.3 with the commands for manipulating storable locks of Chapter 4 and a command for forking a thread:

$$C \quad ::= \quad \dots \mid \mathtt{init}(E) \mid \mathtt{finalise}(E) \mid \mathtt{acquire}(E) \mid \mathtt{release}(E) \mid \mathtt{fork}(\mathtt{f})$$

where `f` is a procedure name. Programs in the language have the same form as those in Chapter 5:

$$\mathtt{f}_0\, \{\mathtt{local}\ \vec{\mathtt{x}}_0\ \mathtt{in}\ C_0\}\ \dots\ \mathtt{f}_l\, \{\mathtt{local}\ \vec{\mathtt{x}}_l\ \mathtt{in}\ C_l\}$$

The procedures $\mathtt{f}_i$ are used to fork threads with $\mathtt{f}_0$ representing the top-level thread. We assume that no thread forks $\mathtt{f}_0$.

As usual, we fix a program S of the above form and represent every procedure body C_i, $i = 0..l$ with its CFG $(N_i, T_i, \mathsf{start}_i, \mathsf{end}_i)$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{init}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\} \cup \{\mathtt{fork}(\mathtt{f}_i) \mid i = 0..l\}$. Let $N =$

$\bigcup_{i=0}^{l} N_i$ and $T = \bigcup_{i=0}^{l} T_i$. We require that the CFGs be deterministic for `fork` commands and let $\mathsf{proc}(v) = k$ if $v \in N_k$.

Note that for simplicity `fork` does not take any parameters to be passed to the thread. In principle, the `fork` command with a parameter $\texttt{fork}(\texttt{f}, E)$ for the procedure declaration

$$\texttt{f}(\texttt{y})\ \{\texttt{local}\ \vec{\texttt{x}}\ \texttt{in}\ C\}$$

can be encoded in our language as

$$\texttt{acquire}(\texttt{l});\ \texttt{z} = E;\ \texttt{fork}(\texttt{f})$$

with the procedure declaration being

$$\texttt{f}\ \{\texttt{local}\ \vec{\texttt{x}}, \texttt{y}\ \texttt{in}\ \texttt{y} = \texttt{z};\ \texttt{release}(\texttt{l});\ C\}$$

Here `z` is a fresh variable and `l` is a pre-initialised lock meant to protect `z`. The lock is used here to ensure mutual exclusion between threads writing parameters to and reading parameters from `z`.

Note also that our language does not include a `join` command for joining a thread, i.e., waiting for its termination. Such a command can be encoded as acquiring a lock passed to the forked thread that it will release before completing.

Encoding parameter passing and thread joining in as described above has its disadvantages (discussed below). However, the minimalistic language we consider here is sufficient to illustrate our main points while keeping the presentation simple. See [33] for native, but more complicated treatment. Proof rules for parameter passing and thread joining could also be derived from the proof rules we present here and the above desugaring.

Model of program states. We consider the separation algebras States, $\text{States}_{\mathbf{c}}$, and $\text{States}_{\mathbf{l}}$ introduced in Sections 4.3 and 4.4 (and the corresponding domains), where we partition program variables into global and local in the spirit of the example model of program states RAM_2 of Section 5.1:

$$\text{Vars} = \text{LocalVars} \uplus \text{GlobalVars}.$$

The part of the stack in a state describing values of local variables is meant to represent the local environment of the procedure running the code of the corresponding thread. We describe local environments with elements of the separation algebra

$$\text{States}_{\mathbf{e}} = \text{LocalVars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms}),$$

with $*$ defined as the $*$ operation on stacks in Section 2.1.1. We denote with $\text{States}^{\mathbf{h}}$, respectively, $\text{States}^{\mathbf{h}}_{\mathbf{c}}$ the subsets of States, respectively, $\text{States}_{\mathbf{c}}$, whose stacks do not contain local variables. We say that $p \in \mathcal{P}(\text{States})^{\top}$ *has an empty environment* if $p \subseteq \mathcal{P}(\text{States}^{\mathbf{h}})^{\top}$.

For a declaration of local variables $\vec{\texttt{x}}$, we define the set of corresponding initial procedure environments $\mathsf{InitEnv}(\vec{\texttt{x}}) \in \mathcal{P}(\text{States}_{\mathbf{e}})$ to be the set of all total functions from the set of variables $\vec{\texttt{x}}$ to $\text{Values} \times \{1\}$. For $i = 0..l$ let $\mathsf{Lve}_i = \mathsf{InitEnv}(\vec{\texttt{x}}_i)$ and $\mathsf{Lv}_i = \mathsf{Lve}_i \times [\,] \times \text{Ints}$.

$$\frac{(v, C, v') \in T \quad C \text{ is not } \mathtt{fork} \quad C, (\eta \uplus s, h, \mathbf{i}) \rightsquigarrow (\eta' \uplus s', h', \mathbf{i})}{\mathsf{pc}[k : v], (g[k : \eta], (s, h, \mathbf{i})) \rightarrow_S \mathsf{pc}[k : v'], (g[k : \eta'], (s', h', \mathbf{i}))}$$

$$\frac{(v, C, v') \in T \quad C \text{ is not } \mathtt{fork} \quad C, (\eta \uplus s, h, \mathbf{i}) \rightsquigarrow \top}{\mathsf{pc}[k : v], (g[k : \eta], (s, h, \mathbf{i})) \rightarrow_S \mathsf{pc}[k : v'], \top}$$

$$\frac{(v, \mathtt{fork}(\mathtt{f}_i), v') \in T \quad \eta' \in \mathsf{Lve}_i \quad (\mathsf{pc}[k : v])(j)\uparrow}{\mathsf{pc}[k : v], (g[k : \eta], (s, h, \mathbf{i})) \rightarrow_S \mathsf{pc}[k : v'][j : \mathsf{start}_i], (g[k : \eta][j : \eta'], (s, h, \mathbf{i}))}$$

Figure 6.1: Operational semantics of programs with storable locks and dynamic thread creation. The relation $\rightsquigarrow$ is defined in Figure 4.7.

Program semantics. Let ThreadIDs $= \{1, 2, \ldots\}$. We define the semantics of the program S by the transition relation

$$\begin{aligned}\rightarrow_S \subseteq \;& ((\text{ThreadIDs} \rightharpoonup_{\text{fin}} N) \times ((\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}}_{\mathbf{c}})) \times \\ & ((\text{ThreadIDs} \rightharpoonup_{\text{fin}} N) \times (((\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}}_{\mathbf{c}}) \cup \{\top\}))\end{aligned}$$

in Figure 5.2, which transforms triples of

- finite partial mappings from thread identifiers to program points (program counters);
- finite partial mappings from thread identifiers to local thread environments; and
- heaps.

An *initial state* of the program S is a state of the form $([1 : \eta_0], \sigma_0)$, where $\eta_0 \in \mathsf{Lve}_0$ and $\sigma_0 \in \text{States}^{\mathbf{h}}_{\mathbf{c}}$. The program S is *safe* when run from an initial state $([1 : \eta_0], \sigma_0)$, if it is not the case that $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \rightarrow^*_S \mathsf{pc}, \top$. We say that a program counter $\mathsf{pc} \in \text{ThreadIDs} \rightharpoonup_{\text{fin}} N$ is *final* if $\forall k.\, \mathsf{pc}(k)\downarrow \Rightarrow \exists i.\, \mathsf{pc}(k) = \mathsf{end}_i$.

6.2 Logic

We extend the logic of Chapter 4 to handle dynamic thread creation as follows. The judgements of the logic are now of the form $\Gamma, I \vdash \{P\}\ C\ \{Q\}$, where Γ is a procedure context (Section 5.2), and I a resource invariant mapping (Section 4.1). We require that the denotations of pre- and postconditions in Γ and resource invariants in I have an empty environment. Let $\mathbf{true_e}$ be the assertion denoting $\text{States}_{\mathbf{e}} \times [\,] \times \text{Ints}$.

A proof of the program S is given by triples

$$\Gamma \vdash \{P * (\vec{\mathtt{x}}_i \Vdash \mathsf{emp}_{\mathbf{h}})\}\ C_i\ \{Q * \mathbf{true_e}\} \text{ for every } \{P\}\ \mathtt{f}_i\ \{Q\} \in \Gamma. \qquad (6.1)$$

The axiom for `fork` is as follows:

$$\frac{}{(\Gamma, \{P\}\ \mathtt{f}_i\ \{Q\}), I \vdash \{P\}\ \mathtt{fork}(\mathtt{f}_i)\ \{\mathsf{emp}\}}\ \textsc{Fork}$$

Upon creating a new thread executing the code of the procedure $\mathtt{f}_i$, the thread that executed `fork` gives up ownership of the precondition of $\mathtt{f}_i$. As in Chapter 4, we consider two variants of the logic—with the conjunction rule and without. In the variant of the logic with the conjunction rule the part of the heap to be transferred to the forked thread at every `fork` command has to be chosen consistently in all branches of the proof. We enforce this by requiring that thread preconditions be precise and annotating each `fork` command with the precondition $P(\vec{X})$ used and the values of all its free logical variables $\vec{X}$, defined by expressions $\vec{E}$ over program variables.[1] The axiom FORK is thus replaced by

$$\frac{}{(\Gamma, \{P(\vec{X})\}\ \mathtt{f}_i\ \{Q\}), I \vdash \{R * P(\vec{X}) \wedge \vec{X} = \vec{E}\}\ \mathtt{fork}_{P(\vec{X}),\vec{E}}(\mathtt{f}_i)\ \{R\}}\ \textsc{Fork}'$$

In the axiom R is meant to supply the permissions for variables needed to evaluate $\vec{E}$.

Aside on annotating `fork`. Note that in some situations it may be too restrictive to require that the values of $\vec{X}$ in FORK′ be defined by expressions that do not contain free logical variables. For example, consider the following application of FORK′:

$$\frac{}{\Gamma, I \vdash \{((\mathtt{u} \Vdash X = \mathtt{u}) * R_1(X)) * P(X) \wedge X = \mathtt{u}\}\ \mathtt{fork}_{P(X),\mathtt{u}}(\mathtt{f}_1)\ \{(\mathtt{u} \Vdash X = \mathtt{u}) * R_1(X)\}} \quad (6.2)$$

where `u` is a local variable of the thread executing the `fork` command, $P(X)$ is $\mathtt{x} \Vdash \mathsf{ls}(\mathtt{x}, X) * \mathsf{ls}(X, \mathtt{NULL})$, `x` is a global variable, $R_1(X)$ is an arbitrary formula that may contain free occurrences of X, and Γ contains $\{P(X)\}\ \mathtt{f}_1\ \{Q_1\}$ for some Q_1. Here the parent thread passes the list pointed to by `x` to its child, while remembering that the list contains the cell at the address `u` (thus, `u` behaves similarly to a cutpoint; see Chapter 5). In this case we are able to annotate `fork` appropriately. However, if the child thread wants to pass the data structure further on via a `fork` command while remembering that it has a node at the address X, it will not be able to provide a corresponding expression to annotate the command with:

$$\frac{}{\Gamma, I \vdash \{R_2(X) * P(X) \wedge X = ?\}\ \mathtt{fork}_{P(X),?}(\mathtt{f}_2)\ \{R_2(X)\}}$$

assuming Γ contains $\{P(X)\}\ \mathtt{f}_2\ \{Q_2\}$ for some Q_2. Such situations can be handled if we add to the logic the auxiliary variable elimination rule [60, 61] for local and global variables, applied last in the proof of a program. Adding the statement $\mathtt{a} = \mathtt{u}$ before the former `fork` command, where `a` is a global auxiliary variable, we can again apply (6.2), but with $P(X)$ defined as $\mathtt{x}, \mathtt{a} \Vdash \mathsf{ls}(\mathtt{x}, X) * \mathsf{ls}(X, \mathtt{NULL}) \wedge X = \mathtt{a}$ and Γ changed accordingly. To handle the latter `fork` command, we can then use the following instance of FORK′ for the newly defined $P(X)$ and Γ:

$$\frac{}{\Gamma, I \vdash \{R_2(X) * P(X) \wedge X = \mathtt{a}\}\ \mathtt{fork}_{P(X),\mathtt{a}}(\mathtt{f}_2)\ \{R_2(X)\}}$$

[1] This can be generalised to the case when $\vec{E}$ depend on the heap.

Thus, the auxiliary variable `a` serves as a witness that the value of the free logical variable X can uniquely be determined from the program state. The fact that we cannot apply the rule EXISTS over a program variable ensures that the same part of the heap is given up at the `fork` command in all branches of the proof. A similar issue comes up in the annotated version of the axiom for the `init` command (INIT$'$, Section 4.1) and can be resolved in the same way.

Example of reasoning. The main thread of the program in Figures 6.2–6.4 allocates an array of `n` objects of the type `DATA` and creates `n` threads to process the objects. It then waits for the termination of all the threads and deallocates the array. We use the desugaring of parameter passing and thread joining presented in Section 6.1. The proof assumes lock sorts P and J (used for parameter passing and thread joining, respectively) with the following invariants:

$$\begin{aligned} I_P(L) &= \mathtt{paramData}, \mathtt{paramJoinLock}, (1/2)\mathtt{paramLock} \Vdash \mathsf{emp}_\mathbf{h}; \\ I_J(L, X) &= \mathsf{emp}_\mathbf{s} \wedge X \mapsto \_. \end{aligned}$$

Soundness. Theorem 4.1 stating the soundness of the logic for storable locks is adapted to our setting as follows.

Theorem 6.1 (Soundness of the logic: variant I). *Consider a proof (6.1) of the program S, where $\{P_0\}\ \mathtt{f}_0\ \{Q_0\} \in \Gamma$ and either*

- *the resource invariants in I and the preconditions of the specifications in Γ are precise, the $*$ operation is cancellative, and* INIT$'$ *and* FORK$'$ *are used instead of* INIT *and* FORK *in the derivation of the triples; or*
- CONJ *and* FORALL *are not used in the derivation of the triples.*

Suppose that $\eta_0 \in \mathsf{Lve}_0$ and a complete state $\sigma_0 \in \mathrm{States}^\mathbf{h}_\mathbf{c}$ is such that for some $W_0 \subseteq$ LockParams

$$\sigma_0 \in \beta\left([\![P_0]\!] * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W_0} [\![I]\!]^\mathsf{F}_A(u, \vec{w})\right)\right).$$

Then the program S is safe when run from $([1 : \eta_0], \sigma_0)$, and whenever $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \rightarrow^*_S \mathsf{pc}, (g, \sigma)$, where pc is final, for some $W \subseteq$ LockParams *we have*

$$\sigma \in \beta\left([\![Q_0]\!] * \left(\mathop{\circledast}_{\{k \ \mid\ \mathsf{pc}(k)\downarrow \wedge k \neq 1\}} \bigsqcup_{\{P\}\ \mathtt{f}_{\mathsf{proc}(\mathsf{pc}(k))}\ \{Q\} \in \Gamma} \mathsf{Exists}(\mathrm{LVars}, [\![Q]\!])\right) * \left(\mathop{\circledast}_{(A,u,\vec{w})\in W} [\![I]\!]^\mathsf{F}_A(u, \vec{w})\right)\right),$$

where Exists *is defined by (5.34).*

```
LOCK *paramLock;
LOCK *paramData;
LOCK *paramJoinLock;
unsigned n;

$\{\mathtt{paramLock}, \mathtt{paramData}, \mathtt{paramJoinLock}, \mathtt{n} \Vdash \mathsf{emp}_{\mathbf{h}}\}$
main() {
  DATA *data;
  int i;
  LOCK **joinLocks;

  $\{O_1 \Vdash \mathsf{emp}_{\mathbf{h}}\}$
  data = new DATA[n];
  joinLocks = new (LOCK*)[n];
  $\{O_1 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_)\}$
  // ...Initialise data...
  $\{O_1 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_)\}$
  paramLock = new LOCK;
  init$_P$(paramLock);
  $\{O_1 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
   $\mathsf{Lock}_P(\mathtt{paramLock}) * \mathsf{Hold}_P(\mathtt{paramLock})\}$
  release(paramLock);
  $\{O_2 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
  for (i = 0; i < n; i++) {
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=0}^{\mathtt{i}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
    joinLocks[i] = new LOCK;
    init$_{J,\mathtt{data}+\mathtt{i}}$(joinLocks[i]);
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=0}^{\mathtt{i}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock}) *$
     $(\exists X. (\mathtt{joinLocks} + \mathtt{i})\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + \mathtt{i}) * \mathsf{Hold}_J(X, \mathtt{data} + \mathtt{i}))\}$
    acquire(paramLock);
    $\{O_1 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=0}^{\mathtt{i}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock}) *$
     $(\exists X. (\mathtt{joinLocks} + \mathtt{i})\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + \mathtt{i}) * \mathsf{Hold}_J(X, \mathtt{data} + \mathtt{i})) *$
     $\mathsf{Hold}_P(\mathtt{paramLock})\}$
```

Figure 6.2: Proof outline for a program with dynamic thread creation (continued in Figures 6.3 and 6.4). Here O_1 is `paramLock`, `paramData`, `paramJoinLock`, `n`, `data`, `i`, `joinLocks` and O_2 is (1/2)`paramLock`, `n`, `data`, `i`, `joinLocks`.

```
    paramData = data+i;
    paramJoinLock = joinLocks+i;
    $\{O_1 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=0}^{\mathtt{i}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock}) *$
     $(\exists X. (\mathtt{joinLocks} + \mathtt{i})\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + \mathtt{i}) * \mathsf{Hold}_J(X, \mathtt{data} + \mathtt{i})) *$
     $\mathsf{Hold}_P(\mathtt{paramLock}) \wedge \mathtt{paramData} = \mathtt{data} + \mathtt{i} \wedge \mathtt{paramJoinLock} = \mathtt{joinLocks} + \mathtt{i}\}$
    fork(process);
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=0}^{\mathtt{i}} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
  }
  $\{O_2 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
  for (i = 0; i < n; i++) {
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=0}^{\mathtt{i}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{i}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=\mathtt{i}}^{\mathtt{n}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
    acquire(joinLocks[i]);
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=0}^{\mathtt{i}} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{i}-1} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock}) *$
     $(\exists X. (\mathtt{joinLocks} + \mathtt{i})\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + \mathtt{i}) * \mathsf{Hold}_J(X, \mathtt{data} + \mathtt{i})) *$
     $\mathsf{Lock}_P(\mathtt{paramLock}))\}$
    finalise(joinLocks[i]);
    delete joinLocks[i];
    $\{O_2 \Vdash \mathtt{i} < \mathtt{n} \wedge (\circledast_{K=0}^{\mathtt{i}} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{i}} (\mathtt{joinLocks} + K)\mapsto\_) *$
     $(\circledast_{K=\mathtt{i}+1}^{\mathtt{n}-1} \exists X. (\mathtt{joinLocks} + K)\mapsto X * \mathsf{Lock}_J(X, \mathtt{data} + K)) * \mathsf{Lock}_P(\mathtt{paramLock}) *$
     $\mathsf{Lock}_P(\mathtt{paramLock}))\}$
  }
  $\{O_2 \Vdash (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{data} + K)\mapsto\_) * (\circledast_{K=0}^{\mathtt{n}-1} (\mathtt{joinLocks} + K)\mapsto\_) * \mathsf{Lock}_P(\mathtt{paramLock})\}$
  delete[n] data;
  delete[n] joinLocks;
  acquire(paramLock);
  finalise(paramLock);
  delete paramLock;
  $\{O_1 \Vdash \mathsf{emp}_{\mathbf{h}}\}$
}
$\{\mathtt{paramLock}, \mathtt{paramData}, \mathtt{paramJoinLock}, \mathtt{n} \Vdash \mathsf{emp}_{\mathbf{h}}\}$
```

Figure 6.3: Proof outline for a program with dynamic thread creation (continued from Figure 6.2, continued in Figure 6.4). Here O_1 is paramLock, paramData, paramJoinLock, n, data, i, joinLocks and O_2 is (1/2)paramLock, n, data, i, joinLocks.

```
$\{\mathtt{paramData}, \mathtt{paramJoinLock}, (1/2)\mathtt{paramLock} \Vdash$
 $\mathtt{paramData} \mapsto \_ * \mathsf{Hold}_P(\mathtt{paramLock}) * \mathsf{Hold}_J(\mathtt{paramJoinLock}, \mathtt{paramData})\}$
process() {
  DATA *data;
  LOCK *joinLock;

  $\{(1/2)\mathtt{paramLock}, \mathtt{paramData}, \mathtt{paramJoinLock}, \mathtt{data}, \mathtt{joinLock} \Vdash$
   $\mathtt{paramData} \mapsto \_ * \mathsf{Hold}_P(\mathtt{paramLock}) * \mathsf{Hold}_J(\mathtt{paramJoinLock}, \mathtt{paramData})\}$
  data = paramData;
  joinLock = paramJoinLock;
  $\{(1/2)\mathtt{paramLock}, \mathtt{paramData}, \mathtt{paramJoinLock}, \mathtt{data}, \mathtt{joinLock} \Vdash$
   $\mathtt{paramData} \mapsto \_ * \mathsf{Hold}_P(\mathtt{paramLock}) * \mathsf{Hold}_J(\mathtt{paramJoinLock}, \mathtt{paramData}) \wedge$
   $\mathtt{data} = \mathtt{paramData} \wedge \mathtt{joinLock} = \mathtt{paramJoinLock}\}$
  release(paramLock);
  $\{\mathtt{data}, \mathtt{joinLock} \Vdash \mathtt{data} \mapsto \_ * \mathsf{Hold}_J(\mathtt{joinLock}, \mathtt{data})\}$
  // ...Process data...
  $\{\mathtt{data}, \mathtt{joinLock} \Vdash \mathtt{data} \mapsto \_ * \mathsf{Hold}_J(\mathtt{joinLock}, \mathtt{data})\}$
  release(joinLock);
  $\{\mathtt{data}, \mathtt{joinLock} \Vdash \mathsf{emp}_{\mathbf{h}}\}$
}
$\{\mathsf{emp}\}$
```

Figure 6.4: Proof outline for a program with dynamic thread creation (continued from Figures 6.2 and 6.3)

We give the proof of the theorem in Section 4.5.1 below. As is the case with Theorem 4.1, this soundness statement does not allow us to prove the absence of memory leaks. This is due to both examples of the kind shown in Figure 4.10 and the fact that the theorem does not bound the number of threads that were forked during the execution of the program. To ensure the absence of memory leaks, we can use the following analogue of Theorem 4.4.

Theorem 6.2 (Soundness of the logic: variant II). *Consider a proof (6.1) of the program S, where $\{P_0\}$ $\mathtt{f}_0$ $\{Q_0\} \in \Gamma$ and the restrictions on the derivations from Theorem 6.1 hold. Suppose further that either Q_0 is intuitionistic, or the postconditions of all the other specifications in Γ are $\mathbf{true_e}$ and the resource invariants in I are admissible. Then for any $\eta_0 \in \mathsf{Lve}_0$ and a complete state $\sigma_0 \in \mathrm{States}_{\mathbf{c}}^{\mathbf{h}}$ such that $\sigma_0 \in \beta\left(\langle [\![P_0]\!] \rangle_{[\![I]\!]}\right)$, the program S is safe when run from $([1 : \eta_0], \sigma_0)$, and whenever $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \to_S^* \mathsf{pc}, (g, \sigma)$, where pc is final, we have $\sigma \in \beta\left(\langle [\![Q_0]\!] \rangle_{[\![I]\!]}\right)$.*

The proof of is similar to that of Theorem 4.4. The requirement that the postcondition of threads be $\mathbf{true_e}$ requires every thread to either dispose of the thread-local data structures before completing, or to transfer their ownership to other threads using locks. Admissibility of resource invariants then ensures that all the data structures in the final state are accounted for. Note that this theorem does not allow us to prove the absence of memory leaks due to undisposed system resources associated with threads: in practice, these are typically disposed of only when the thread is joined, and our programming language does not provide a `join` command. See [33] for an alternative formulation of the language and the logic that allows detecting such memory leaks.

6.2.1 Proof of soundness

To prove Theorem 6.1, we combine the notions of semantic proofs of Sections 4.5.1 and 5.2.2. Given a set of specification indices $\mathcal{R}$ and a set of procedures $\{\mathtt{f}_0, \dots, \mathtt{f}_l\}$, here we define a semantic proof as a tuple $(C, G, \mathcal{E}, \mu, \mathcal{I})$, where

- C is a command with a CFG $(N, T, \mathsf{start}, \mathsf{end})$ over the set of primitive commands $\mathsf{Seq} \cup \{\mathtt{init}(E), \mathtt{init}_{A,\vec{F}}(E), \mathtt{finalise}(E), \mathtt{acquire}(E), \mathtt{release}(E)\} \cup \{\mathtt{fork}(\mathtt{f}_i) \mid i = 0..l\} \cup \{\mathtt{fork}_{P(\vec{X}),\vec{E}}(\mathtt{f}_i) \mid i = 0..l\}$;
- $G : N \to D_{\mathfrak{l}}$ maps program points of C to semantic annotations;
- $\mathcal{E}$ is a semantic procedure context of the form
$$\mathcal{E} = \big\{\{p_i^j\}\ \mathtt{f}_i\ \{q_i^j\} \mid i = 0..l,\ j \in \mathsf{specs}(i)\big\}, \tag{6.3}$$
where $\mathsf{specs}(i) \subseteq \mathcal{R}$ and $p_i^j, q_i^j \in D_{\mathfrak{l}}$ have an empty environment;
- $\mu : N \times \mathcal{R} \to D_{\mathfrak{l}}$ is a mapping such that for an edge $(v, \mathtt{f}_i, v') \in T$, $\mu(v', j)$ gives the frame for the jth specification of the procedure forked at v;
- $\mathcal{I} \in \text{InvMaps}$ is a semantic resource invariant mapping

such that for all edges $(v, C', v') \in T$

- (4.5)–(4.8) hold for the corresponding commands C';
- if C' is $\mathtt{fork}(\mathtt{f}_i)$, then
$$G(v) \sqsubseteq \bigsqcup_{j \in \mathsf{specs}(i)} p_i^j * \mu(v', j) \tag{6.4}$$
and
$$G(v') \sqsupseteq \bigsqcup_{j \in \mathsf{specs}(i)} \mu(v', j). \tag{6.5}$$
- if C' is $\mathtt{fork}_{P(\vec{X}),\vec{E}}(\mathtt{f}_i)$, then
$$G(v') \neq \top \Rightarrow G(v) \neq \top \wedge \forall (s, h, \mathbf{i}) \in G(v).\, \exists \vec{u}.\ [\![\vec{E}]\!]_s = \vec{u} \wedge (s, h, \mathbf{i}) \in G(v') * \mathsf{Subst}([\vec{X} : \vec{u}], [\![P(\vec{X})]\!]), \tag{6.6}$$
where Subst is defined by (5.35).

Similarly to Section 5.2.2, inequalities (6.4) and (6.5) represent a semantic counterpart of the axiom FORK closed under the applications of the rules FRAME and DISJ. In fact, the inequalities are a particular case of (5.13) and (5.14), where the postcondition of the procedure is empty. Inequality (6.6) is a counterpart of the axiom FORK$'$. We treat the annotated version of `fork` separately to record the fact that the precondition and the values of logical variables are chosen according to the annotation.

We define the functions

$$\nu_k : \text{States} \to ((\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}}),\ k \in \text{ThreadIDs};$$
$$\nu_0 : \text{States}^{\mathbf{h}} \to ((\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}})$$

as follows:

$$\nu_k(s, h, \mathbf{i}) = ([k : \eta], (s', h, \mathbf{i})),\ k \in \text{ThreadIDs}; \quad \nu_0(s, h, \mathbf{i}) = ([\,], (s, h, \mathbf{i})),$$

where $\eta \in \text{States}_{\mathbf{e}}$ and $s' \in \text{States}^{\mathbf{h}}$ are such that $\eta \uplus s' = s$. Let the operation $\star$ on $(\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}}$ be defined as follows: $(g_1, \xi_1) \star (g_2, \xi_2) = (g_1 \uplus g_2, \xi_1 * \xi_2)$. We also use pointwise liftings of ν_k, ν_0, and $\star$ and an iterated version of the lifted $\star$. Additionally, we lift the function β defined in Section 4.4 to $\mathcal{P}((\text{ThreadIDs} \rightharpoonup_{\text{fin}} \text{States}_{\mathbf{e}}) \times \text{States}^{\mathbf{h}})^{\top}$ in the obvious way.

Lemma 6.3 (Soundness of the intermediate interpretation). *Consider a semantic procedure context $\mathcal{E}$ of the form (6.3) for some set of indices $\mathcal{R}$. Assume semantic proofs $(C_i, G_i^j, \mathcal{E}, \mu(i,j), \mathcal{I})$ for $i = 0..l$, $j \in \mathsf{specs}(i)$, where $G_i^j(\mathsf{start}_i) \sqsupseteq p_i^j * \mathsf{Lv}_i$ and take $j_1 \in \mathsf{specs}(0)$. If $\eta_0 \in \mathsf{Lve}_0$ and a complete state $\sigma_0 \in \text{States}_{\mathbf{c}}^{\mathbf{h}}$ are such that for some $W_0 \subseteq \text{LockParams}$*

$$\{([1 : \eta_0], \sigma_0)\} \sqsubseteq \beta\left(\nu_1(G_0^{j_1}(\mathsf{start}_0)) \star \nu_0\left(\circledast_{(A,u,\vec{w}) \in W_0} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right),$$

then whenever $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \to_S^* \mathsf{pc}, \zeta$, for some $W \subseteq \text{LockParams}$ and $\rho :$ $\text{ThreadIDs} \to \mathcal{R}$ such that

$$\rho(1) = j_1 \wedge \forall k.\, \mathsf{pc}(k){\downarrow} \Rightarrow \rho(k) \in \mathsf{specs}(\mathsf{proc}(\mathsf{pc}(k))),$$

we have

$$\{\!|\zeta|\!\} \sqsubseteq \beta\left(\left(\bigstar_{\{k \ \mid\ \mathsf{pc}(k)\downarrow\}} \nu_k(G_{\mathsf{proc}(\mathsf{pc}(k))}^{\rho(k)}(\mathsf{pc}(k)))\right) \star \nu_0\left(\circledast_{(A,u,\vec{w}) \in W} \mathcal{I}_A^{\mathsf{F}}(u, \vec{w})\right)\right).$$

Proof. We prove the lemma by induction on the length of the derivation of ζ in the operational semantics of the program S. We consider only the case of the `fork` command, since the others are dealt with in the same way as in the proof of Lemma 4.5. Thus, assume

$$[1 : \mathsf{start}_1], ([1 : \eta_0, \sigma_0]) \to_S^* \mathsf{pc}[t : v], (g, \sigma) \to_S \mathsf{pc}[t : v'][j : \mathsf{start}_i], (g[j : \eta], \sigma),$$

$(v, \texttt{fork}(\texttt{f}_i), v') \in T$, $\eta \in \mathsf{Lve}_i$, and $(\mathsf{pc}[t : v])(j)\uparrow$. Let $\mathsf{proc}(v) = u$. Suppose that for some $W \subseteq \text{LockParams}$ and $\rho : \text{ThreadIDs} \to \mathcal{R}$ such that $\rho(1) = j_1$ and $\forall k.\, \mathsf{pc}(k)\downarrow \Rightarrow \rho(k) \in \mathsf{specs}(\mathsf{proc}(\mathsf{pc}(k)))$, we have

$$\{(g, \sigma)\} \sqsubseteq \beta(\nu_t(G_u^{\rho(t)}(v)) \star r), \tag{6.7}$$

where

$$r = \left(\mathop{\circledast}_{\{k \ \mid \ \mathsf{pc}(k)\downarrow \wedge k \neq t\}} \nu_k(G^{\rho(k)}_{\mathsf{proc}(\mathsf{pc}(k))}(\mathsf{pc}(k))) \right) \star \nu_0 \left(\mathop{\circledast}_{(A,u,\vec{w}) \in W} \mathcal{I}^{\mathsf{F}}_A(u, \vec{w}) \right).$$

It is sufficient to show that for some $w \in \mathsf{specs}(i)$

$$\{(g[j : \eta], \sigma)\} \sqsubseteq \beta(\nu_t(G_u^{\rho(t)}(v)) \star \nu_j(G_i^w(\mathsf{start}_i)) \star r).$$

From (6.4) and (6.7) we get

$$\{(g, \sigma)\} \sqsubseteq \beta\left(\nu_t\left(\bigsqcup_{w \in \mathsf{specs}(i)} p_i^w * \mu(u, \rho(t), v', w)\right) \star r\right).$$

Hence, for some $w \in \mathsf{specs}(i)$

$$\{(g, \sigma)\} \sqsubseteq \beta(\nu_t(p_i^w * \mu(u, \rho(t), v', w)) \star r).$$

By (6.5), we then get

$$\{(g, \sigma)\} \sqsubseteq \beta(\nu_t(p_i^w * G_u^{\rho(t)}(v')) \star r).$$

Since $\eta \in \mathsf{Lve}_i$, $G_i^w(\mathsf{start}_i) \sqsupseteq p_i^w * \mathsf{Lv}_i$ and p_i^w has an empty environment, from this we get

$$\begin{aligned}\{(g[j : \eta], \sigma)\} &\sqsubseteq \beta(\nu_t(p_i^w * G_u^{\rho(t)}(v')) \star \nu_j(\mathsf{Lv}_i) \star r)\\ &\sqsubseteq \beta(\nu_t(G_u^{\rho(t)}(v')) \star \nu_j(\mathsf{Lv}_i * p_i^w) \star r)\\ &\sqsubseteq \beta(\nu_t(G_u^{\rho(t)}(v')) \star \nu_j(G_i^w(\mathsf{start}_i)) \star r)\end{aligned}$$

as required. □

Proof of Theorem 6.1. Consider the proof (6.1) of the program S. Let $\mathcal{E}$ be the semantic procedure context of the form (6.3) constructed out of Γ.

Consider first the case of the logic with the conjunction rule. It is easy to show that analogues of Lemmas 3.8 and 3.11 hold for semantic proofs of the form introduced in this section where all `init` and `fork` commands are annotated. This allows us to show that from a proof of $\Gamma, I \vdash \{P * (\vec{\mathtt{x}}_i \Vdash \mathsf{emp}_\mathbf{h})\}\ C_i\ \{Q * \mathbf{true_e}\}$, we can construct a semantic proof $(C_i, G, \mathcal{E}, \mu, [\![I]\!])$ for some G and μ such that $G(\mathsf{start}_i) = [\![P]\!] * \mathsf{Lv}_i$, $G(\mathsf{end}_i) \sqsubseteq [\![Q * \mathbf{true_e}]\!]$ $\forall v.\, G(v) \sqsubset \top$, and $\forall v, j.\, \mu(v, j) \sqsubset \top$. The set of semantic proofs constructed in this way from (6.1) satisfies the conditions of Lemma 6.3, which implies Theorem 6.1.

Consider now the case of the logic without the conjunction rule. In this case, we can establish an analogue of Lemma 5.11. The statement of Theorem 6.1 is then reduced to Lemma 6.3 in the same way as in the proof of the soundness of EXISTS in Section 5.5.1.□

6.3 Deriving the analysis

We assume the setting of Section 4.6 and all the ingredients for the analysis defined there. The analysis for our programming language is a simple extension of the one in Figure 4.11 that accounts for the `fork` command. The analysis processes it by splitting the state of the current thread into two parts, one of which is transferred to the newly created thread, and the other stays with the parent. As in the case of the interprocedural analysis of Chapter 5, the splitting may introduce cutpoints. We therefore additionally parameterise our analysis with ingredients similar to the ones used in the interprocedural analysis of Section 5.5.2:

- The abstract state $\mathsf{Lv}_i^\sharp \in D_{\mathrm{I}}^\sharp$, $i = 0..l$ represents the initial environment of the procedure $\mathtt{f_i}$:
$$\mathsf{Lv}_i \sqsubseteq \gamma(\mathsf{Lv}_i^\sharp).$$
- The functions $\mathsf{Cutpoints}_i : D_{\mathrm{I}}^\sharp \to \mathcal{P}(\mathrm{LVars})$, $i = 0..l$ such that $\mathsf{Cutpoints}_i(p)$ returns the set of logical variables to represent cutpoints arising in the state p when the procedure $\mathtt{f}_i$ is forked. Let $\mathsf{Cutpoints}(\top) = \emptyset$.
- The functions $\mathsf{Child}_i : D_{\mathrm{I}}^\sharp \to D_{\mathrm{I}}^\sharp$ and $\mathsf{Parent}_i : D_{\mathrm{I}}^\sharp \to D_{\mathrm{I}}^\sharp$ for $i = 0..l$ determine the splitting of the abstract state when the procedure $\mathtt{f}_i$ is forked: Child_i determines the part of the state that is passed to the newly created thread and Parent_i the part that stays with its parent. We require that they split the state soundly:
$$\forall p \in D_{\mathrm{I}}^\sharp.\, \gamma(p) \sqsubseteq \mathsf{Exists}(\mathsf{Cutpoints}_i(p), \gamma(\mathsf{Child}_i(p)) * \gamma(\mathsf{Parent}_i(p))). \tag{6.8}$$
We also require that $\gamma(\mathsf{Child}_i(p))$ have an empty environment.

Let $\mathsf{pre} \in D_{\mathrm{I}}^\sharp$ be an abstract element representing the set of initial states of the program S such that $\gamma(\mathsf{pre})$ has an empty environment. The analysis operates on the domain $\widehat{D}^\sharp = (N \to D_{\mathrm{I}}^\sharp) \times \mathrm{InvMaps}^\sharp$ and is defined by the functional $\mathcal{F}^\sharp(\mathsf{pre}) : \widehat{D}^\sharp \to \widehat{D}^\sharp$ in Figure 6.5.

A heuristic for determining heap splittings. When the program being analysed is desugared from a program with `fork` passing parameters to the newly created thread, we can define the functions Child_k and Parent_k similarly to the functions $\mathsf{ProcLocal}_k$ and Frame_k in the interprocedural analysis: the part of the heap reachable from the global variables holding the actual parameters, along with the necessary permissions for the lock protecting the parameters, is passed to the newly created thread (as described in Section 6.1 and illustrated by the example in Figure 6.2). A typical pattern occurring in code with dynamic thread creation is for several worker threads to synchronise access to a shared data structure using a lock passed to them by the same parent thread. To handle such situations, the straightforward implementation of Child_k described above has to be

$\mathcal{F}^\sharp(\mathsf{pre})(G^\sharp, I^\sharp) = (\widetilde{G}^\sharp, \widetilde{I}^\sharp)$, where

- $\widetilde{G}^\sharp(\mathsf{start}_0) = \mathsf{pre} *^\sharp \mathsf{Lv}_0^\sharp$;
- $\widetilde{G}^\sharp(\mathsf{start}_i) = \bigsqcup_{(v, \mathtt{fork}(\mathtt{f}_i), v') \in T} \mathsf{Child}_i(G^\sharp(v))$, $i = 1..l$;
- $\widetilde{G}^\sharp(v') = \bigsqcup_{(v,C,v') \in T} g_C^\sharp(G^\sharp(v))$
  for every program point $v' \in N \backslash \{\mathsf{start}_k \mid k = 0..l\}$, where

$$g_C^\sharp(p) = \begin{cases} f_C^\sharp(p), & \text{if } C \in \mathsf{Seq} \cup \{\mathtt{init}(E), \mathtt{finalise}(E)\}; \\ f_C^\sharp(I^\sharp, p), & \text{if } C \text{ is } \mathtt{acquire}(E); \\ \mathsf{ThreadLocal}_E(p), & \text{if } C \text{ is } \mathtt{release}(E); \\ \mathsf{Parent}_i(p), & \text{if } C \text{ is } \mathtt{fork}(\mathtt{f}_i); \end{cases}$$

- $\widetilde{I}_A^\sharp = \bigsqcup_{(v, \mathtt{release}(E), v') \in T} \mathsf{Protected}_E(G^\sharp(v), A)$ for every lock sort $A \in \mathcal{A}$.

Figure 6.5: Thread-modular analysis for concurrent programs with storable locks and dynamic thread creation

modified to pass only a fraction of the permission for every lock in the part of the heap reachable from the actual parameters.

Since the postcondition of a thread in our desugaring of the `join` command is passed to another thread via the invariant of the lock used to signal the thread's termination, the analysis has to initialise the lock appropriately, assigning a separate lock sort to every symbolic heap representing a possible postcondition, and parameterising the invariant with the values of cutpoints appearing in the thread's precondition. The treatment of `init` commands in the analysis is flexible enough to allow this. A realistic analysis implementation would treat parameter passing and thread joining natively.

Soundness. The soundness of the analysis is stated by the following two theorems, analogous to Theorems 4.1 and 4.4, which we can use in the cases when we are interested in detecting memory leaks or not, respectively.

Theorem 6.4 (Soundness of the analysis: variant I). *Let $(G^\sharp, I^\sharp)$ be a fixed point of the functional $\mathcal{F}^\sharp(\mathsf{pre})$. If $\eta_0 \in \mathsf{Lve}_0$ and a complete state $\sigma_0 \in \mathrm{States}_{\mathbf{c}}^{\mathbf{h}}$ are such that for some $W_0 \subseteq \mathrm{LockParams}$*

$$\{([1:\eta_0], \sigma_0)\} \sqsubseteq \beta\left(\nu_1(\gamma(\mathsf{pre} *^\sharp \mathsf{Lv}_0^\sharp)) \star \nu_0\left(\circledast_{(A,u,\vec{w}) \in W_0} (\gamma(I^\sharp))_A^{\mathsf{F}}(u, \vec{w})\right)\right),$$

then whenever $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \to_S^* \mathsf{pc}, \zeta$*, for some* $W \subseteq \mathrm{LockParams}$*, we have*

$$\{\!|\zeta|\!\} \sqsubseteq \beta \left(\nu_1(\gamma(G^\sharp(\mathsf{pc}(1)))) \star \left(\bigoplus_{\{k \ \mid \ \mathsf{pc}(k)\downarrow \wedge k \neq 1\}} \nu_k(\mathsf{Exists}(\mathrm{LVars}, \gamma(G^\sharp(\mathsf{pc}(k))))) \right) \star \right.$$
$$\left. \nu_0 \left(\circledast_{(A,u,\vec{w}) \in W} (\gamma(I^\sharp))_A^{\mathsf{F}}(u, \vec{w}) \right) \right).$$

Let us lift the notion of closure (Definition 4.2) to $\mathcal{P}((\mathrm{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathrm{States}_{\mathbf{e}}) \times \mathrm{States}^{\mathbf{h}})^\top$ as follows: for all $\mathcal{I} \in \mathrm{InvMaps}$, $\langle \top \rangle_{\mathcal{I}} = \top$ and for all $p \in \mathcal{P}((\mathrm{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathrm{States}_{\mathbf{e}}))$

$$\langle p \rangle_{\mathcal{I}} = \bigsqcup \left\{ \{g\} \times \langle \{\sigma\} \rangle_{\mathcal{I}} \mid (g, \sigma) \in p \right\},$$

where the later occurrence of $\langle \cdot \rangle_{\mathcal{I}}$ corresponds to the notion of closure in Definition 4.2.

Theorem 6.5 (Soundness of the analysis: variant II). *Let* $(G^\sharp, I^\sharp)$ *be a fixed point of the functional* $\mathcal{F}^\sharp(\mathsf{pre})$*,* $\eta_0 \in \mathsf{Lve}_0$*, and a complete state* $\sigma_0 \in \mathrm{States}_{\mathbf{c}}^{\mathbf{h}}$ *be such that*

$$\{\sigma_0\} \sqsubseteq \beta\left(\langle \gamma(\mathsf{pre}) \rangle_{\gamma(I^\sharp)} \right).$$

If $[1 : \mathsf{start}_0], ([1 : \eta_0], \sigma_0) \to_S^* \mathsf{pc}, \zeta$*, and either* $\gamma(G^\sharp(\mathsf{pc}(k)))$ *is intuitionistic for some* k *or the resource invariants in* $\gamma(I^\sharp)$ *are admissible, then*

$$\{\!|\zeta|\!\} \sqsubseteq \beta \left(\left\langle \nu_1(\gamma(G^\sharp(\mathsf{pc}(1)))) \star \left(\bigoplus_{\{k \ \mid \ \mathsf{pc}(k)\downarrow \wedge k \neq 1\}} \nu_k(\mathsf{Exists}(\mathrm{LVars}, \gamma(G^\sharp(\mathsf{pc}(k))))) \right) \right\rangle_{\gamma(I^\sharp)} \right).$$

Note that under the conditions of the theorem when $\gamma(G^\sharp(\mathsf{end}_i)) \sqsubseteq [\![\mathbf{true_e}]\!]$ for $i = 1..l$ and pc is final, we have

$$\{\!|\zeta|\!\} \sqsubseteq \beta \left(\left\langle \nu_1(\gamma(G^\sharp(\mathsf{end}_0))) \star \left(\bigoplus_{\{k \ \mid \ \mathsf{pc}(k)\downarrow \wedge k \neq 1\}} \nu_k([\![\mathbf{true_e}]\!]) \right) \right\rangle_{\gamma(I^\sharp)} \right),$$

which allows us to check the absence of memory leaks by examining $G^\sharp(\mathsf{end}_0)$. We can check admissibility using the approximate criterion given in Section 4.6.

6.4 Related work

Most existing program logics for concurrent programs consider parallel composition, a less general programming construct than dynamic thread creation. The logics for storable locks we discussed in Section 4.7 can also reason about dynamic thread creation. The relationship of these logics to ours is as discussed in Section 4.7.

Feng and Shao [28] presented a rely-guarantee logic for reasoning about concurrent assembly code with dynamic thread creation. They do not have an analogue of our rule for ownership transfer at `fork` commands. On a higher level, our logic for threads relates to theirs in the same way as separation logic relates to rely-guarantee reasoning: the former is good at describing ownership transfer, and the latter at describing interference.

Dodds et al. [25] have recently extended the logic proposed here to reason about interference more flexibly. In their logic, assertions can contain tokens permitting or prohibiting threads from modifying the shared state in a certain way. One can transfer the ownership of such a token to a newly created thread in its precondition, thus allowing the thread to modify the shared state as described by the token, and get the token back upon the thread's termination, thus ensuring that the environment does not modify the shared state after that.

Chapter 7

Conclusion

In this dissertation we have proposed logics and analyses for the verification of concurrent heap-manipulating programs. The logics and the analyses avoid the problem of reasoning about all thread interleavings in concurrent programs using modular reasoning, which results in tractable proofs and efficient tools. They also handle realistic programming constructs, including those not addressed by prior work: storable locks, procedures, and dynamically-created threads. Thus, our results provide the necessary ingredients for building practical verification tools.

We have further shown that reasoning principles for concurrent heap-manipulating programs one might want to use in manual and automatic reasoning differ in subtle ways: all our logics come with non-standard variants that exclude the conjunction rule in exchange for lifting the restrictions that are hard for automatic analyses to satisfy. In this way, our work on program analyses has led to further insights into program logics.

Our results come with several caveats. First, while considering realistic programming constructs, in this dissertation we did not tie ourselves to a particular concurrency library so as not to clutter presentation. A specialisation of our logics to a particular programming language or an implementation of our analyses has to fill in the missing details. Second, we have mostly developed separate logics and analyses for every programming construct considered, showing their compositions only in the cases of non-trivial interactions between the constructs. This has an advantage that reasoning principles for every construct are presented in a clean way. Composing all logics and analyses presented here (e.g., to handle a language with storable locks, procedures, and threads) can be done as illustrated in Section 5.4. Finally, throughout this dissertation we assumed a sequentially consistent memory model, whereas modern processors and programming languages exhibit weak memory models (see, e.g., [51, 59]). Most of these models, however, come with a theorem establishing that a program satisfying a suitably specialised notion of data-race freedom assuming a sequentially consistent memory model is executed on the weak memory model as though the memory were sequentially consistent. Since all our logics and analyses ensure data-race freedom, it should not be problematic to use them to reason about programs running on weak memory models.

We conclude by noting some directions of further research that our results suggest.

7.1 Future work

A unified logic for concurrent heap-manipulating programs. In this dissertation we restricted ourselves to programs that satisfy O'Hearn's Ownership Hypothesis, i.e., mostly to coarse-grained programs. As we discussed in Section 4.7, handling fine-grained and non-blocking concurrency requires methods different from ours. Modern programs use a mixture of synchronisation techniques, hence, one needs to unified logic to reason about them. Developing such a logic is ongoing work [26, 25, 23].

Liveness properties. We have restricted ourselves to verifying safety properties. Developing methods for verifying liveness properties of concurrent heap-manipulating programs has been the subject of our ongoing work [35, 18]. Note that safety properties verified by our analyses are needed to support liveness proofs.

Other concurrency constructs. We have restricted ourselves to programs with shared memory that use locks for synchronisation. We conjecture that our methods can be also applied to shared-memory programs using monitors and message passing. On the other hand, reasoning about distributed programs without shared memory will probably require different approaches.

More robust invariant inference algorithms. As we discussed in Section 3.6, the use of a fixed heuristic in the thread-modular analysis to decide the splitting of the heap at `release` commands is not always satisfactory. It is certainly worth investigating more principled approaches to resource invariant inference (Calcagno et al. [14] have made a promising step in this direction).

Getting the conjunction rule back. The conjunction rule is useful for combining the results of two proofs or two analyses. For example, the reduced product construction [55] in abstract interpretation effectively uses it. Since resource invariants inferred by the thread-modular analysis of Chapter 3 may be imprecise, one may wonder if there are other ways to ensure the soundness of the conjunction rule other than requiring precision. Intuitively, for the conjunction rule to be sound, proofs or runs of the analyses being combined have to split the state in the same way at every `release` command in the program. One way to enforce this for program analyses is to require that the analyses being combined use the same heuristic defined in the concrete domain, i.e., satisfying (3.6) and (3.7). This requirement can be enforced in the logic by adding a side condition to the global version of the axiom RELEASE ensuring that the state is split according to the heuristics $\mathsf{ThreadLocal}_k$ and $\mathsf{Protected}_k$. It is possible to prove the soundness of a reduced product construction for the analysis of Chapter 3, and the soundness of the corresponding logic including CONJ and the global versions of ACQUIRE and RELEASE, but excluding FRAME. Unfortunately, the logic with both CONJ and FRAME is unsound, since the function $\mathsf{ThreadLocal}_k$ defined using reachability from the entry points is not local: if a part of the heap reachable from the entry points is hidden using FRAME and then reattached to the new local state, the result will be incompatible with just applying RELEASE, since in this case the part will be given up. Thus, an analogue of Lemma 3.8(iii) does not hold in this case. The situation is clearly unsatisfactory, since we need FRAME to process procedure calls in a local way (Section 5.4). There are several possible approaches to restoring the soundness of FRAME, such as requiring it not to frame out a part of the heap that may be transferred to the protected state by $\mathsf{Protected}_k$ at a `release` command inside the code FRAME is being applied to. Ultimately, the appropriate solution seems to depend on the concrete setting we are using the logic or the analysis in.

Bibliography

[1] M. Barnett, B. E. Chang, R. Deline, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05: Symposium on Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006. (Cited on page 95.)

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07: Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007. (Cited on pages 37 and 38.)

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05: Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005. (Cited on page 64.)

[4] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006. (Cited on page 32.)

[5] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV'08: Conference on Computer Aided Verification*, volume 5123 of *LNCS*, pages 399–413. Springer, 2008. (Cited on page 66.)

[6] A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research, 1989. (Cited on page 10.)

[7] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: Symposium on Principles of Programming Languages*, pages 259–270. ACM, 2005. (Cited on pages 17, 69, 70, and 73.)

[8] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06: Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006. (Cited on page 32.)

[9] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02: Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM, 2002. (Cited on page 66.)

[10] J. Boyland. Checking interference with fractional permissions. In *SAS'03: Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003. (Cited on pages 17 and 70.)

[11] S. D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *MFPS'06: Conference on Mathematical Foundations of Programming Semantics*, volume 158 of *ENTCS*, pages 123–150, 2006. (Cited on page 28.)

[12] S. D. Brookes. A semantics of concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007. Preliminary version appeared in *CONCUR'04: Conference on Concurrency Theory.* (Cited on pages 28, 54, and 119.)

[13] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09: Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009. (Cited on page 132.)

[14] C. Calcagno, D. Distefano, and V. Vafeiadis. Compositional resource invariant synthesis. In *APLAS'09: Asian Symposium on Programming Languages and Systems.* Springer, 2009. To appear. (Cited on pages 68 and 150.)

[15] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS'07: Symposium on Logic in Computer Science*, pages 366–378. IEEE, 2007. (Cited on pages 15, 20, 28, and 105.)

[16] C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS'07: Static Analysis Symposium*, volume 4634 of *LNCS*, pages 233–248. Springer, 2007. (Cited on page 95.)

[17] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI'02: Conference on Programming Languages Design and Implementation*, pages 258–269. ACM, 2002. (Cited on pages 42, 51, and 66.)

[18] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *POPL'07: Symposium on Principles of Programming Languages*, pages 265–276. ACM, 2007. (Cited on pages 10 and 150.)

[19] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, pages 106–130. Dunod, 1976. (Cited on pages 32 and 33.)

[20] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977. (Cited on pages 29 and 31.)

[21] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comp.*, 2(4):511–547, 1992. (Cited on page 38.)

[22] A. de Bruin. Goto statements: Semantics and deduction systems. *Acta Inf.*, 15:385–424, 1981. (Cited on page 54.)

[23] T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Parkinson. Providing a fiction of disjoint concurrency. Unpublished manuscript, 2009. (Cited on pages 96 and 150.)

[24] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06: Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006. (Cited on pages 33, 34, and 35.)

[25] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP'09: European Symposium on Programming*, volume 5502 of *LNCS*, pages 363–377. Springer, 2009. (Cited on pages 96, 147, and 150.)

[26] X. Feng. Local rely-guarantee reasoning. In *POPL'09: Symposium on Principles of Programming Languages*, pages 315–327. ACM, 2009. (Cited on pages 96 and 150.)

[27] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP'07: European Symposium on Programming*, volume 4421 of *LNCS*. Springer, 2007. (Cited on pages 69, 95, and 96.)

[28] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *ICFP'05: International Conference on Functional Programming*, pages 254–267. ACM, 2005. (Cited on page 146.)

[29] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI'00: Conference on Programming Languages Design and Implementation*, pages 219–232. ACM, 2000. (Cited on page 66.)

[30] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03: Workshop on Model Checking Software*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003. (Cited on page 49.)

[31] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS'06: Static Analysis Symposium*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006. (Cited on pages 13, 98, 129, and 131.)

[32] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS'07: Asian Symposium on Programming Languages and Systems*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007. (Cited on pages 12 and 71.)

[33] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. Technical Report MSR-TR-2007-39, Microsoft Research, 2007. (Cited on pages 12, 134, and 141.)

[34] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI'07: Conference on Programming Languages Design and Implementation*, pages 266–277. ACM, 2007. (Cited on pages 12, 52, 66, and 68.)

[35] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL'09: Symposium on Principles of Programming Languages*, pages 16–28. ACM, 2009. (Cited on pages 10, 32, and 150.)

[36] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI'03: Workshop on Types in Languages Design and Implementation*, pages 13–25. ACM, 2003. (Cited on page 66.)

[37] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *PLDI'07: Conference on Programming Languages Design and Implementation*, pages 256–265. ACM, 2007. (Cited on page 132.)

[38] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In *APLAS'08: Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008. (Cited on page 95.)

[39] J. Hayman and G. Winskel. Independence and concurrent separation logic. In *LICS'06: Symposium on Logic in Computer Science*, pages 147–156. IEEE, 2006. (Cited on page 28.)

[40] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008. (Cited on page 78.)

[41] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. (Cited on page 15.)

[42] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on the Semantics of Algorithmic Languages*, volume 188 of *LNM*, pages 102–116. Springer, 1971. (Cited on pages 99 and 131.)

[43] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP'08: European Symposium on Programming*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008. (Cited on page 95.)

[44] IEEE. *IEEE Standard 1003.1-2001.* 2001. (Cited on pages 10 and 69.)

[45] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01: Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001. (Cited on pages 15, 20, 26, and 87.)

[46] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP'05: European Symposium on Programming*, volume 3444 of *LNCS*, pages 124–140. Springer, 2005. (Cited on page 53.)

[47] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP'09: European Symposium on Programming*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009. (Cited on page 95.)

[48] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV'06: Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 547–561. Springer, 2006. (Cited on pages 38 and 50.)

[49] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS'07: Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 3–18. Springer, 2007. (Cited on pages 38 and 50.)

[50] R. Manevich, T. Lev-Ami, G. Ramalingam, M. Sagiv, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS'08: Static Analysis Symposium*, volume 5079 of *LNCS*. Springer, 2008. (Cited on pages 66 and 67.)

[51] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL'05: Symposium on Principles of Programming Languages*, pages 378–391. ACM, 2005. (Cited on page 149.)

[52] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07: Symposium on Principles of Programming Languages*, pages 327–338. ACM, 2007. (Cited on page 66.)

[53] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06: Conference on Programming Languages Design and Implementation*, pages 308–319. ACM, 2006. (Cited on page 66.)

[54] G. C. Necula. Proof-carrying code. In *POPL'97: Symposium on Principles of Programming Languages*, pages 106–119. ACM, 1997. (Cited on page 53.)

[55] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 2005. (Cited on page 150.)

[56] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. Preliminary version appeared in *CONCUR'04: Conference on Concurrency Theory.* (Cited on pages 10, 20, 27, 29, 54, 60, 67, 68, and 69.)

[57] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01: Workshop on Computer Science Logic*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. (Cited on page 25.)

[58] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):1–50, 2009. Preliminary version appeared in *POPL'04: Symposium on Principles of Programming Languages.* (Cited on pages 20, 68, and 104.)

[59] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs'09: Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009. (Cited on page 149.)

[60] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. (Cited on page 136.)

[61] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976. (Cited on page 136.)

[62] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS'06: Symposium on Logic in Computer Science*, pages 137–146. IEEE, 2006. (Cited on pages 17, 19, 26, 99, and 132.)

[63] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *PLDI'06: Conference on Programming Languages Design and Implementation*, pages 320–331. ACM, 2006. (Cited on pages 42, 51, 66, and 94.)

[64] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95: Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995. (Cited on pages 98, 112, and 131.)

[65] W. Retert and J. Boyland. Interprocedural analysis for JVML verification. In *FTfJP'02: Workshop on Formal Techniques for Java-like Programs*, 2002. Technical Report NIII-R0204, Computing Science Department, University of Nijmegen. (Cited on page 68.)

[66] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. (Cited on pages 15, 26, and 64.)

[67] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL'05: Symposium on Principles of Programming Languages*, pages 296–309, 2005. (Cited on pages 97, 98, 99, 114, and 131.)

[68] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *CC'01: Conference on Compiler Construction*, volume 2027 of *LNCS*, pages 133–149. Springer, 2001. (Cited on page 132.)

[69] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Technical Report 26, Tel-Aviv University, 2004. (Cited on pages 99, 112, and 131.)

[70] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS'05: Static Analysis Symposium*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005. (Cited on pages 98, 99, and 131.)

[71] P. Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Softw.*, 3(6):46–57, 1986. (Cited on page 10.)

[72] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. (Cited on pages 39 and 66.)

[73] S. Savage, M. Burrows, G. Nelson, P. Soblvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Comp. Syst.*, 15(4):371–411, 1997. (Cited on pages 42, 51, and 66.)

[74] S. Seo, H. Yang, and K. Yi. Automatic construction of Hoare proofs from abstract interpretation results. In *APLAS'03: Asian Symposium on Programming Languages and Systems*, volume 2895 of *LNCS*, pages 230–245. Springer, 2003. (Cited on page 63.)

[75] V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. Technical Report UCAM-CL-TR-726, University of Cambridge Computer Laboratory, 2008. (Cited on pages 95 and 96.)

[76] V. Vafeiadis. RGSep action inference. In *VMCAI'10: Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2010. To appear. (Cited on pages 66 and 95.)

[77] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07: Conference on Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007. (Cited on pages 66, 68, 69, 95, and 96.)

[78] M. Y. Vardi. Verification of concurrent programs—the automata-theoretic framework. *Ann. Pure Appl. Logic*, 51:79–98, 1991. (Cited on page 32.)

[79] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993. (Cited on page 106.)

[80] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL'01: Symposium on Principles of Programming Languages*, pages 27–40. ACM, 2001. (Cited on pages 41 and 66.)

[81] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV'08: Conference on Computer Aided Verification*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008. (Cited on pages 64, 98, 99, and 131.)

[82] H. Yang, O. Lee, C. Calcagno, D. Distefano, and P. W. O'Hearn. On scalable shape analysis. Technical Report RR-07-10, Queen Mary, 2007. (Cited on page 131.)

[83] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *FOSSACS'02: Conference on Foundations of Software Science and Computation Structures*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002. (Cited on page 21.)