Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic

Clément Hurlin

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

présentée et soutenue par

Clément HURLIN

Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic

Thèse dirigée par Marieke HUISMAN

soutenue le 14 septembre 2009

Jury:

Mr. Lars BIRKEDAL	Professeur	Rapporteur
Mr. Frank PIESSENS	Professeur	Rapporteur
Mr. Claude MARCHÉ	Directeur de recherche	Président
Mme. Tamara REZK	Chargé de recherche	
Mme. Marieke HUISMAN	Professeur assistant	Directrice de thèse

Résumé

Cette thèse développe (1) un système de vérification en logique de séparation pour des programmes à la Java parallèles et (2) de nouvelles analyses utilisant la logique de séparation.

La logique de séparation est une variante de la logique linéaire [50] qui connaît un récent succès pour la vérification de programmes [93, 84, 88]. Dans la littérature, la logique de séparation a été appliquée à des programmes *while* simples [93], des programmes while parallèles [84], et des programmes objets séquentiels [88]. Ces remarques sont développées dans l'introduction (chapitre 1). Dans cette thèse nous continuons ces travaux en adaptant la logique de séparation aux programmes objets parallèles similaires à Java.

Dans ce but, nous développons de nouvelles règles de vérification pour les primitives de Java concernant le parallélisme. Pour se faire, nous élaborons un langage modèle similaire à Java (chapitre 2) qui sert de base aux chapitres ultérieurs. Ensuite, nous mettons en œuvre un système de vérification en logique de séparation pour les programmes écrits dans ce langage modèle (chapitre 3).

Le chapitre 4 présente des règles de vérification pour les primitives *fork* et *join*. La primitive fork démarre un nouveau thread et la primitive join permet d'attendre la terminaison d'un thread. Ces primitives sont utilisées par un certain nombre de langages: C++, python, et C♯. Le chapitre 5 décrit des règles de vérification pour les verrous réentrants qui sont utilisés en Java. Les verrous réentrants – contrairement aux verrous Posix – peuvent être acquis plusieurs fois. Cette propriété simplifie la tâche du programmeur mais complique la vérification.

La suite de la thèse présente trois nouvelles analyses utilisant la logique de séparation. Au chapitre 7, nous continuons le travail de Cheon et al. sur les séquences d'appels de méthodes autorisées (aka *protocoles*) [32]. Nous étendons ces travaux aux programmes parallèles et inventons une technique afin de vérifier que les spécifications des méthodes d'une classe sont conformes au protocole de cette classe. Au chapitre 8, nous développons un algorithme pour montrer qu'une formule de logique de séparation n'en implique *pas* une autre. Cet algorithme fonctionne de la manière suivante: étant donné deux formules A et B, la taille des modèles potentiels de A et B est calculée approximativement, puis; si les tailles obtenues sont incompatibles, l'algorithme conclut que A n'implique pas B. Cet algorithme est utile dans les vérificateurs de programmes car ceux-ci doivent souvent décider des implications. Enfin, au chapitre 9, nous montrons comment paralléliser et optimiser des programmes en observant leurs preuves en logique de séparation. Cet algorithme est implanté dans l'outil *éterlou*.

Pour finir, au chapitre 10, nous concluons et proposons des pistes pour compléter les résultats de la présente thèse.

Mots Clés

Spécification de programmes, vérification de programmes, logique de séparation, orienté-objet, parallélisme, Java, parallélisation automatique.

Abstract

This thesis develops a verification system in separation logic for multithreaded Java programs. In addition, this thesis shows three new analyses based on separation logic.

Separation logic is a variant of linear logic [50] that did a recent breakthrough in program verification [93, 84, 88]. In the literature, separation logic has been applied to simple while programs [93], while programs with parallelism [84], and sequential object oriented programs [88]. We complete these works by adapting separation logic to multithreaded object-oriented programs la Java.

To pursue this goal, we develop new verification rules for Java's primitives for multithreading. The basis of our work consists of a model language that we use throughout the thesis (Chapter 2). All our formalisation is based on this model language.

First, we propose new verification rules for fork and join (Chapter 4). Fork starts a new thread while join waits until the receiver thread terminates. Fork and join are used in many mainstream languages including Java, C++, C#, and python. Then, we describe verification rules for reentrant locks i.e. Java's locks (Chapter 5). Reentrant locks - as opposed to Posix locks - can be acquired multiple times (and released accordingly). This property eases the programmer's task, but it makes verification harder.

Second, we present three new analyses based on separation logic. The first analysis (Chapter 7) completes Cheon et al.'s work on sequences of allowed method calls (i.e. protocols). We extend this work to multithreaded programs and propose a technique to verify that method contracts comply with class protocols. The second analysis permits to show that a separation logic formula does not entail another formula (Chapter 8). This algorithm works as follows: given two formulas A and B, the size of A and B's models is approximated; then if the sizes of models are incompatible, the algorithm concludes that A does not entail B. This algorithm is useful in program verifiers, because they have to decide entailment often. The third analysis shows how to parallelize and optimize programs by looking at their proofs in separation logic (Chapter 9). This algorithm is implemented in a tool called terlou.

Finally, we conclude and discuss possible future work in Chapter 10.

Keywords

Program specification, program verification, separation logic, object-oriented, multithreading, Java, automatic parallelization.

Remerciements

Avant toute chose, je remercie mon amoureuse. Merci pour tes constants encouragements, surtout ces derniers mois : ton amour inconditionnel, malgré la distance, voilà ce qui me pousse. Toi et moi, c'est une thèse sans conclusion...

Ces premières lignes sont aussi dédiées à mes parents, qui ne me tiennent pas rancune quand je ne donne pas signe de vie ; mais qui sont toujours là quand j'ai besoin d'eux, et ce, dans tous les domaines.

Scientifiquement, mes plus profonds remerciements vont à Christian Haack et Marieke Huisman. Le premier car il a su trier le bon grain de l'ivraie dans mes divagations, il m'a enseigné la rigueur nécessaire pour écrire des preuves de qualité, et il m'a démontré l'importance d'avoir des formalisations claires comme de l'eau de roche. La seconde pour m'avoir appris à écrire des articles, tout était à faire, mais grâce à sa patience et à sa diplomatie, j'ai énormément progressé.

Je remercie également ceux qui m'ont entouré aussi bien scientifiquement que socialement. À ce titre, je remercie Gilles Barthe, Yves Bertot, Patrice Chalin, Amir Hossein Ghamarian, Joe Kiniry, Gustavo Petri, Erik Poll, Loïc Pottier, Arend Rensink, Tamara Rezk, Laurence Rideau, et Laurent Théry. Mentions spéciales à Benjamin Grégoire, rare combinaison de marcels, de maîtrise du poulailler, et d'humour ; et Anne Pacalet pour la prodigualité de ses conseils et nos nombreuses discussions. Je n'oublie pas ceux avec qui j'ai passé des moments au boulot *et* en dehors. Nicolas Julien pour son incroyable perspicacité à travailler juste ce qu'il faut, Radu Grigore, Sylvain Heraud, Alex Summers, et #linux. Merci aussi à celles qui ont un pied dans ce monde et – heureusement – un pied dans la réalité : Nathalie Bellesso et Joke Lammerink.

Enfin, il est des influences qui résonnent en moi comme les plus belles lignes de basse. Leurs mots ou leurs choix m'ont guidé : Jacques Ellul, Bernard Charbonneau, et ma *princesse*.

Contents

Chapter 1

Introduction

L'homme ne peut décider de suivre telle voie plutôt que la voie technique: ou bien il décide d'user du moyen traditionnel ou personnel et alors ses moyens ne sont pas efficaces, ils seront étouffés ou éliminés, ou bien il décide d'accepter la nécessité technique, il vaincra soumis de façon irrémédiable à l'esclavage technique. Il n'y a donc absolument aucune liberté de choix.

Le système technicien (1977)
Jacques Ellul

1.1 Motivation

Software is ubiquitous in a variety of activities. Withdrawing cash from an ATM, waiting for the level crossing to go up at Enschede's station, flying a plane, watching TV: in all these situations, you interact with software, or worse your *integrity* relies on software. The impact of software failure can vary, though. If your TV works incorrectly, you might just switch it off and get it fixed. When you rely on software, however, a failure can be deadly: if the level crossing does not go down when a train arrives you might get crushed. Consequently, making sure that software works correctly is – in some fields – required. This could be for good reasons: ensuring that trains, cars, and planes are safe. This could be for terrible reasons: ensuring that tanks and fighter jets work correctly.

The goal of software verification is to provide some confidence in software. From a high-level point of view, software verification is a set of methods that take programs as input, and outputs certain guarantees about the input programs. Such guarantees include "this program does not perform unexpected operations", "this program never dereferences a null pointer", "this program allocates that amount of memory", "this program computes an answer to the problem X", "this program does not transfer money", "this program does not leak my passwords" etc. The level of confidence provided by software verification is inversely proportional to the effort required to apply the technique used. Entirely automatic techniques give guarantees such as "this program does not perform unexpected operations". Providing the guarantee that "this program never dereferences a null pointer" is less likely to be automatic. Showing that "this program computes an answer to the problem X" usually requires human interaction. Typically, the human interaction is to tune the verification technique (with annotations for example). In this thesis, we show a technique that requires high effort from programmers, but provides high confidence.

One of the first verification technique was type checking. Type checking ensures that programs only perform valid operations. For example, the program below cannot be type checked:

```
int n := 0;
string s := "tutut";
return (n = s);
```

This program is invalid, because it does not make sense: it compares an integer (`n`) and a string (`s`). It is good practice to consider this an error (although some languages might allow such a behavior). That is exactly the role of type checking. Considering the trade off between automation and the level of confidence provided, type systems are entirely automatic, but provide a low level of confidence. For example, consider the following program:

```
zeroify(Integer o){
 o.val := 0;
}
```

The program above might crash if the `Integer` object passed to `zeroify` is null. In this case the assignment `o.val := 0` fails. Here, Java's or C++'s type systems do not discover this possible error[1]. Basic type checkers answer to questions such as "does this program perform unexpected operations ?" and advanced type checkers answer to questions such as "does this program dereference a null pointer". Type systems, however, fail to ensure advanced properties. For example consider the following piece of C code:

```
int queens(int a, int b, int c){
 int d=0,e=a& b& c,f=1;
 if(a)for(f=0;d=(e-=d)&-e;f+=queens(a-d,(b+d)*2,(c+d)/2));
 return f;
}
main(q){
 scanf("%d",&q);
 printf("%d\n",t(~(~0<<q),0,0));
}
```

This mysterious piece of code (studied by Filliâtre [47]) computes the number of solutions to the well-known n-queens problem, where n is read on standard input. Interestingly, this program is actually very efficient at solving this problem. Type systems cannot be used to verify that this program *really* solves the n-queen problem. To verify this program, full-fledged software verification aka *static checking* is necessary. Generally speaking, static checking requires programmers to *specify* (or *annotate*) programs, i.e., to write down what programs are supposed to do. Static checkers take specified programs as input and output mathematical formulas called *proof obligations*. If all proof obligations can be discharged, then the input program satisfies the specifications. If some proof obligations cannot be proven, then either the prover is too weak, or specifications are wrong, or the program is wrong.

Static checking answers basic questions such as "does this program dereference a null pointer ?" and advanced questions such as "does this program compute an answer to the problem X ?". On one hand, static checking requires effort from programmers, because it requires programs to be annotated. On the other hand, static checking ensures strong guarantees on software. For example of static checking's powerfulness, Filliâtre proved that the program above does solve the n-queens problem.

[1] Advanced type systems [59] could catch this bug, but they are not yet integrated in mainstream languages.

Static checking offers a level of confidence according to the programmer's annotations. If the programmer specifies solely memory safety (i.e., absence of null dereferences), then static checking will provide only this level of confidence. Consequently, static checking is not tailored to a particular problem. That is why better techniques might exist for specific problems. Such problems include "does my program transfer money ?" or "does my program leaks my passwords ?". For these problems, techniques focusing on security and privacy [10] perform better than static checking.

In the next section, we give detailed explanations on static checking. Even if specific techniques might perform better than static checking for some problems, static checking is the basis of program verification. Static checking's advantage is that it lets the programmer specify what property he wants to verify. Consequently, it handles a large variety of problems. Below, we review techniques related to static checking and position our work w.r.t. to these techniques.

1.2 Background

Hoare [57] first started program verification as we know it today. He proposed reasoning with triples consisting of two formulas and a program. Such triples (which are now called "Hoare triples") have the following form:

$$\{F\}c\{G\}$$

Above, F and G are the two formulas and c is the program (c should be read as "command"). This triple has the following meaning: if program c starts in a state described by formula F, then, if c terminates, it will terminate in a state described by G. In Hoare triples, F is called the *precondition* and G is called the *postcondition*. Early work focused on how to build such triples for large programs. For small programs, building Hoare triples is easy. For example, the following Hoare triple formalizes the meaning of variable assignment:

$$\{F[E/x]\}x := E\{F\}$$

Above $F[E/x]$ is formula F where all free occurrences of variable x have been replaced by expression E. Then, the rule should be read as follows: what was true about E before the assignment (i.e., $F[E/x]$) is true for x after the assignment (i.e., F). Hoare triples for large programs are built by using special rules. For example, the rule below shows how to build a triple for a program that successively executes program c and program c':

$$\frac{\{F\}c\{G\} \qquad \{G\}c'\{H\}}{\{F\}c;\ c'\{H\}}\ (\text{Seq})$$

In this rule the two triples above the bar are called the *premises*, while the triple below the bar is called the *conclusion*. This rule (called the sequence rule) means the following: given that program c satisfies the triple $\{F\}c\{G\}$ and that program c' satisfies the triple $\{G\}c'\{H\}$, then it can be deduced that program $c;\ c'$ satisfies the triple $\{F\}c;c'\{H\}$.

A limitation of traditional Hoare logic is that it has no way to express what objects are accessed and what objects are *not* accessed by method calls. For example consider the following Java-like class:

```
class 2DPoint{
 int x,y;
 requires true; ensures this.x == newx;
 void setX(int newx){ this.x = newx; }
}
```

Above, method `setX` is annotated with a pre- and a postcondition. Preconditions indicate what client must establish before calling a method, while postconditions indicate what methods ensure to clients after they return. Here, the precondition `true` is vacuous: it imposes nothing to clients. The postcondition `this.x == newx` indicates that method `setX` updates field `x` with the value passed as a parameter.

Method `setX`'s precondition illustrates that Hoare logic is inadequate to express what objects are accessed and what objects are not accessed by method calls. This precondition is not enough informative. First, it does not indicate that field `x` of the receiver is accessed. Second, it does not indicate which fields are not accessed (that is, all fields of all objects except field `x` of the receiver). Knowing what fields are accessed and what fields are not accessed is crucial to verify programs. If these informations are available, the algorithms used for verification become simpler and program verification itself is easier.

This problem (which is called the *frame* or the *aliasing* problem) has been thoroughly studied in the last decade. Recently, in the field of program verification, two techniques arose to deal with this problem.

The first approach was to incorporate ownership techniques [34, 81] into program verifiers [41, 28, 6]. In this approach, programmers have to specify an ownership relation between objects. Roughly, ownership relations enforce that objects are *only* modified by their owners (which are also objects). This permits to tame aliasing, because strong assumptions about possible modifications can be made. A shortcoming of these approaches is that they do not help to verify multithreaded programs. First, because ownership relations are static (once they are established they can never be changed), they are too restrictive for multithreaded programs where the owners of objects change dynamically. For example, an object might be owned successively by different threads or an object can be successively owned by a thread and then by a lock. Second, ownership relations suffer from the shortcoming that they should obey a fixed discipline (contrary to being specified by programmers). Consequently, programs that fail complying to the discipline cannot be verified.

The second approach used to deal with aliasing was to use a different logic, that allows to express ownership relations dynamically. In most program verifiers [6, 28, 9], formulas are written in some variant of *first order logic*. Recently, a new logic called *separation logic* appeared [93]. One novelty of this logic is that it encompasses both ownership and a solution to aliasing. The basic formula of separation logic is the *points-to* predicate $x.f \mapsto v$. This formula has a dual meaning: firstly, it asserts that the object field $x.f$ contains data value v and, secondly, it represents a permission to access the field $x.f$ (i.e., it expresses ownership of $x.f$). Separation logic also features a new operator: the separating conjunction $\star$. Formula $F \star G$ means that formula F holds for a part of the heap, while formula G holds for a *separate* part of the heap. This operator is the key to tame aliasing. In separation logic, class `2DPoint` is specified as follows:

```
class 2DPoint{
 int x,y;
 requires this.x ↦ _; ensures this.x ↦ newx;
 void setX(int newx){ this.x = newx; }
}
```

Now, the precondition expresses what field should be owned by callers to `setX`: simply `this.x`. In addition, it implicitly expresses that everything that does not appear in the precondition is left untouched, because there is no permission to access it.

Until now, we have only discussed sequential programs. Parallel programs (which are also called *multithreaded* or *concurrent* programs), however, are nowadays becoming more and more important. First, because of the high demand on software; programmers have to write fast (scientific computing), responsive (user interfaces), and reliable code. Second, as the speed of processors stopped increasing quickly, hardware manufacturer started increasing the number of processors on motherboards: nowadays, most laptops feature 2 or 4 processors. Consequently, to efficiently use the hardware's capabilities, programmers now have to write multithreaded programs. This is not an easy task though: writing multithreaded programs is much more difficult than writing sequential programs.

First, one has to decide how to parallelize to achieve the desired result. This requires to use special patterns (worker threads, divide and conquer algorithms etc.) and to implement them correctly. Second, and more concretely, one must avoid *data races*. A data race occurs when two threads write to the same location in the heap simultaneously, or when one thread writes to a location and another thread simultaneously reads the same location. Data races are rarely intended and can result in unexpected behaviors [61]. Third, one has to avoid *deadlocks*. A deadlock occurs when a thread t_0 waits for another thread t_1 to perform some action, while t_1 also waits for t_0 to perform some action. In this case, both t_0 and t_1 are blocked forever, i.e., deadlocked.

Because of the issues sketched above, verifying multithreaded programs requires dedicated techniques that go beyond a simple adaptation of the verification techniques for sequential programs. Historically, the first verification technique for parallel programs was proposed by Owicki and Gries [87]. They advocated to check the interference between commands from different processes. This solution, however, suffered from two defects. First, it was non-modular: if a new process was added to the system, one had to check possible interferences of this new process with *all* other processes. Second, Owicki and Gries's technique had a high complexity because *all* program points of a process had to be checked for interferences against *all* program points of other processes.

Later, Jones proposed rely-guarantee [72] to tame the complexity of verification of parallel programs. In this framework, programmers have to annotate each process with a rely condition (what this process assumes about the environment, i.e., the other processes) and a guarantee condition (what this process guarantees to the other processes). While this method is flexible (it handles fine-grained parallelism), it had the disadvantage that the rely and the guarantee condition of a process spread into all proofs of this process. Concretely, at each program point, a process must make sure that it does not break its guarantee condition.

In the last years, O'Hearn [84] adapted separation logic to reason about parallel programs. He discovered that separation logic solves both the modularity problem and the

complexity problem of Owicki and Gries and Jones. Indeed, verification of concurrent programs with separation logic is *modular*: each thread can be verified separately. Further, complexity of verification is acceptable, because interferences between threads are only handled at synchronization points.

O'Hearn studied simple while programs with a parallel operator $\|$ and locks. He developed the following Hoare rule to deal with the parallel operator:

$$\frac{\{F\}C\{F'\} \qquad \{G\}C'\{G'\}}{\{F \star G\}C\|C'\{F' \star G'\}} \text{ (Parallel)}$$

Side condition: C does not modify any variables free in G, C', and G' (and vice versa)

The rule (Parallel) can be understood as follows: for two threads to execute in parallel, they should access disjoint parts of the heap (i.e., F and G). When the two threads terminate, the resulting heap is simply the $\star$ conjunction of the heaps obtained after both threads's execution (i.e., F' and G').

While the rule (Parallel) is sound, it is too restrictive: it forbids two threads to simultaneously read a location. Boyland [23] solved this problem with a very intuitive idea, which was later adapted to separation logic [21]. The idea is that (1) *access tickets are splittable*, (2) *a split of an access ticket still grants read access* and (3) *only a whole access ticket grants write access*. To account for multiple splits, Boyland uses fractions, hence the name *fractional permissions*. In permission-accounting separation logic [21], access tickets $x.f \mapsto v$ are superscripted by fractions π. $x.f \overset{\pi}{\longmapsto} v$ is equivalent to $x.f \overset{\pi/2}{\longmapsto} v \ * \ x.f \overset{\pi/2}{\longmapsto} v$. In the Hoare rules, writing requires the full fraction 1, whereas reading just requires *some* fraction π:

$$\{x.f \overset{1}{\longmapsto} \_\}x.f = v\{x.f \overset{1}{\longmapsto} v\} \qquad \{x.f \overset{\pi}{\longmapsto} v\}y = x.f\{x.f \overset{\pi}{\longmapsto} v \ * \ v == y\}$$

Permission-accounting separation logic maintains the global invariant that the sum of all fractional permissions to the same cell is always at most 1. This prevents read-write and write-write conflicts, but permits concurrent reads.

O'Hearn also presented Hoare rules for locks in separation logic, where he elegantly adapted an old idea from concurrent programs with shared variables [3]: Each lock is associated with a *resource invariant* which describes the part of the heap that the lock guards. When a lock is acquired, it lends its resource invariant to the acquiring thread. Dually, when a lock is released, it takes back its resource invariant from the releasing thread. This is formally expressed by the following Hoare rules:

$$\frac{I \text{ is } x\text{'s resource invariant}}{\{\texttt{true}\}x.\texttt{lock()}\{I\}} \qquad \frac{I \text{ is } x\text{'s resource invariant}}{\{I\}x.\texttt{unlock()}\{\texttt{true}\}}$$

Because of O'Hearn's work, it has been known for a while that separation logic was a convenient and powerful tool to reason about parallel programs.

Another extension to separation logic was Parkinson's work on an object-oriented language à la Java. He proposed to use *abstract predicates* to hide details of implementations. Abstract predicates are definitions that are *transparent* to class implementors, but *opaque* to clients. In addition, Parkinson showed how to use abstract predicates to deal with subtyping.

This thesis extends separation logic to deal with *multithreaded* object-oriented programs à la Java. The next section details this thesis's contributions.

1.3 This Thesis

We build on the work of O'Hearn on concurrent separation logic [84] and on the work of Parkinson on object-oriented separation logic [88]. Here, we describe at a high-level the contributions of this thesis.

Extension of Separation Logic to Multithreaded Java: We extend separation logic to a multithreaded Java-like language. This is a *challenge*, because Java's primitives for multithreading differ from the primitives studied by O'Hearn. First, Java uses fork and join to dynamically create threads instead of the idealized parallel operator $\|$. Fork and join are harder to model than the $\|$ operator because they are more general: $\|$ can be encoded using fork and join (but not vice versa). Second, Java uses reentrant locks (i.e., locks that can be acquired twice or more) instead of O'Hearn's non-reentrant locks.
This extension of separation logic is *important*, because it permits to verify realistic Java programs. A crucial achievement is that we prove that our extension of separation logic is *sound* (with pen and paper).

New Analyzes Based on Separation Logic: The first analysis extends method specifications with *protocol* specifications and permits to check that method specifications are correct. Again, this is important, because writing correct specifications is difficult: specifications are harder to test (runtime checking is required) and tool support for debugging specifications is scarce. The second analysis shows how to disprove entailment between separation logic formulas. The crucial feature of this algorithm is that it has a *low complexity* i.e., it provides a quick way to disprove entailment (this is useful in program verifiers). The third analysis allows to parallelize programs that have been proven correct with separation logic. As it is harder to write parallel programs than sequential programs, this analysis provides programmers an alternative solution to obtain correct parallel programs. We provide a *high degree of confidence* that these analyzes are correct: the soundness of the second analysis has been checked with the Coq theorem prover [36], while the soundness of the third analysis has been checked on paper.

Overview. We now give an outline of the thesis and present the contributions of each chapter.

From Chapter 2 to Chapter 6, we show a verification system that handles fundamental aspects of multithreaded Java-like programs.

- In Chapter 2, we present a Java-like language and its semantics. While this is not new, this is a basis for the following chapters.
- In Chapter 3, we present how to specify and verify Java-like programs in separation logic. To do this, we present a set of Hoare rules that we have shown sound. The key contributions of this chapter are: for expressiveness, we use parameterized classes and methods, we present a subtyping relation that avoids reverification of inherited methods, and to export useful facts to clients, we use axioms and `public` modifiers.
- In Chapter 4, we show how to specify and verify Java-like programs with fork and join as concurrency primitives. The key contributions of this chapter are verification rules for fork and join. These verification rules are expressed with general purpose abstract predicates, groups, and class axioms. Further, these verification rules allow both single write-joiner and multiple readonly-joiner. Like Chapters 2 and 3, the

work from this chapter has been done in close collaboration with Christian Haack. It is published in the International Conference on Algebraic Methodology and Software Technology (AMAST 2008) [53].

- In Chapter 5, we show how to specify and verify Java-like programs with reentrant locks (i.e., locks that can be acquired twice or more). The key contributions of this chapter are: verification rules for reentrant locks, a way to express resource invariants (including inheritance) with general purpose abstract predicates, and an example of how the combination of our expressive type system and separation logic handles a challenging lock-coupling algorithm. The work from this chapter has been done in collaboration with Christian Haack and Marieke Huisman. It is published in the Asian Symposium on Programming Languages and Systems (APLAS 2008) [53].
- In Chapter 6, we prove that the verification system defined from Chapters 3 to 5 is sound. This is a crucial achievement to obtain a good confidence in this system.

In Chapters 7, 8, and 9, we present three new analyzes based on separation logic.

- In Chapter 7, we specify class protocols. The key contributions of this chapter are: an extension of Cheon et al.'s work to multithreaded Java-like programs that use parameterized classes and advanced protocols, and a new technique to check that method contracts are correct w.r.t. protocols. The work presented in this chapter is published in the ACM Symposium on Applied Computing (SAC 2009) [64].
- In Chapter 8, we present a new technique to disprove entailment between separation logic formulas. We abstract formulas by the sizes of their possible models and use comparisons of these sizes to disprove entailment between formulas. Intuitively, to disprove that formula F entails formula G, it suffices to show that there exists a model (i.e., a heap) of F whose size is smaller than the size of all models of G. We give two different ways of calculating sizes of models, which have differing complexity and precision. This work has been done in collaboration with François Bobot and Alexander J. Summers. It is published in the proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO 2009) [65].
- In Chapter 9, we show a new technique to automatically parallelize and optimize programs by rewriting their proofs. Proofs are represented as a derivation of Hoare triples. The core of the procedure uses separation logic's (Frame) rule to statically detect parts of the state which are useless for a command to execute. Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, late lock acquirement, and improvement of temporal locality [96]. Optimizations are expressed as rewrite rules between proof trees and are performed automatically. We present an implementation of the procedure. The work from this chapter is published in the International Symposium on Static Analysis (SAS 2009) [63].

Finally, we conclude and discuss future work in Section 10.

Chapter 2

A Java-like Language

This chapter presents the Java-like language that will serve as the basis for Chapter 3, 4, 5, 6, and 7. This language contains the core object-oriented features of Java: mutable fields, inheritance and method overriding, and interfaces. It does not contain, however, static fields, super calls, and reflexion.

This chapter is structured as follows: In Section 2.1 we show the syntax of an object-oriented model language à la Java, in Section 2.2 we present the semantics of our model language, and in Section 2.3 we discuss related formalization of Java-like languages. We do not prove a type soundness theorem for this language in this chapter. Such a theorem will be included in a preservation theorem in Chapter 6.

Notes:

(a) The work from this chapter has been done in collaboration with Christian Haack. It served at the basis for a publication in the International Conference on Algebraic Methodology and Software Technology (AMAST 2008) [53].

(b) To help the reader, the symbol index in Appendix C on page 187 recapitulates where symbols used in Chapters 2 to 7 are introduced.

2.1 Syntax

This section formally presents the Java-like language that is used to write programs and specifications. The language distinguishes between read-only variables $\imath$, read-write variables ℓ, and logical variables α. The distinction between read-only and read-write variables is not essential, but often avoids the need for syntactical side conditions in the proof rules (see Section 3.3). Method parameters (including `this`) are read-only; read-write variables can occur everywhere else, while logical variables can only occur in specifications and types. Apart from this distinction, the *identifier domains* are standard:

Identifier Domains:

$C, D \in \mathsf{ClassId}$	class identifiers (including `Object`)
$I, J \in \mathsf{IntId}$	interface identifiers
$s, t \in \mathsf{TypeId} = \mathsf{ClassId} \cup \mathsf{IntId}$	type identifiers
$o, p, q, r \in \mathsf{ObjId}$	object identifiers
$f \in \mathsf{FieldId}$	field identifiers

$m \in \mathsf{MethId}$	method identifiers
$P \in \mathsf{PredId}$	predicate identifiers
$\imath \in \mathsf{RdVar}$	read-only variables (including `this`)
$\ell \in \mathsf{RdWrVar}$	read-write variables
$\alpha \in \mathsf{LogVar}$	logical variables
$x, y, z \in \mathsf{Var} = \mathsf{RdVar} \cup \mathsf{RdWrVar} \cup \mathsf{LogVar}$	variables

The distinction between read-only and read-write variables means that programs written in this language are not valid Java programs. However, this is not a real restriction because any valid Java program can be translated directly in our syntax.

Values are integers, booleans, object identifiers, and `null`. For convenience, read-only variables can be used as values directly. Read-only and read-write variables can only contain these basic values. In this chapter, *specifications values* only range over logic variables and values, but they will be extended later.

$$n \in \mathsf{Int} \qquad u, v, w \in \mathsf{Val} ::= \texttt{null} \mid n \mid b \mid o \mid \imath$$
$$b \in \mathsf{Bool} = \{\texttt{true}, \texttt{false}\} \qquad \pi \in \mathsf{SpecVal} ::= \alpha \mid v$$

Now we define the *types* used in our language. Since interfaces and classes (defined next) can be parameterized with specification values, object types are of the form $t\texttt{<}\bar{\pi}\texttt{>}$.

$$T, U, V, W \in \mathsf{Type} ::= \texttt{void} \mid \texttt{int} \mid \texttt{bool} \mid t\texttt{<}\bar{\pi}\texttt{>}$$

Next, *class declarations* are defined. As in Java, classes can `ext`(end) other classes, and `impl`(ement) interfaces. Classes declare *fields*, *abstract predicates*, *class axioms*, and *methods*. Abstract predicates and class axioms are part of our specification language and are explained in Section 3.2.1. Methods have pre/postcondition specifications, parameterized by logical variables. The meaning of a specification is defined via a universal quantification over these parameters. In examples, we usually leave the parameterization implicit, but it is treated explicitly in the formal language.

Class Declarations:

$F \in \mathsf{Formula}$	specification formulas (see Section 3.2)
$spec ::= \texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ F\texttt{;}$	pre/postconditions
$fd ::= T\ f\texttt{;}$	field declarations
$pd ::= \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\texttt{;}$	predicate definitions (see Section 3.2.1)
$ax ::= \texttt{axiom}\ F\texttt{;}$	class axioms (see Section 3.2.1)
$md ::= \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}$	methods (scope of $\bar{\alpha}, \bar{\imath}$ is $\bar{T}, spec, U, \bar{V}, c$)
$cl \in \mathsf{Class} ::= \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}$	class (scope of $\bar{\alpha}$ is $\bar{T}, U, \bar{V}, fd\texttt{*}, pd\texttt{*}, ax\texttt{*}, md\texttt{*}$)

In a similar way, we define *interfaces*. In method types mt, the receiver is made explicit by separating it from parameters by ";". This is convenient for later developments.

Interface Declarations:

$pt ::= \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>;}$	predicate types
$mt ::= \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}V_0\ \imath_0\texttt{;}\ \bar{V}\ \bar{\imath}\texttt{)}$	method types (scope of $\bar{\alpha}, \bar{\imath}$ is $\bar{T}, spec, U, V_0, \bar{V}$)
$int \in \mathsf{Interface} ::= \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$	

interfaces (scope of $\bar{\alpha}$ is $\bar{T}, \bar{U}, pt\texttt{*}, ax\texttt{*}, mt\texttt{*}$)

Class and interface declarations define *class tables*: $ct \subseteq \mathsf{Interface} \cup \mathsf{Class}$. We use the symbol $\preceq_{ct}$ for the order on type identifiers induced by class table ct. We often leave the subscript ct implicit. We impose the following sanity conditions on ct: (1) $\preceq_{ct}$ is antisymmetric, (2) if t (except `Object`) occurs anywhere in ct then t is declared in ct and (3) ct does not contain duplicate declarations or a declaration of `Object`. We write $\mathsf{dom}(ct)$ for the set of all type identifiers declared in ct.

Subtyping is inductively defined by the following rules:

Subtyping $T <: T$:

$$T <: T \qquad T <: U, U <: V \;\Rightarrow\; T <: V \qquad s\texttt{<}\bar{T}\,\bar{\alpha}\texttt{> ext } t\texttt{<}\bar{\pi}'\texttt{>} \;\Rightarrow\; s\texttt{<}\bar{\pi}\texttt{>} <: t\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}$$
$$t\texttt{<}\bar{\pi}\texttt{>} <: \texttt{Object} \qquad t\texttt{<}\bar{T}\,\bar{\alpha}\texttt{> impl } I\texttt{<}\pi'\texttt{>} \;\Rightarrow\; t\texttt{<}\bar{\pi}\texttt{>} <: I\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}$$

Commands are sequences of head commands hc and local variable declarations, terminated by a return value:

Commands:

$$\begin{array}{rcl} op & \supseteq & \{\texttt{==}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{\, C \texttt{ classof}\,\} \cup \{\, \texttt{instanceof } T\} \\ c \in \mathsf{Cmd} & ::= & v \mid T\ \ell;\ c \mid T\ \imath\texttt{=}\ell;\ c \mid hc;\ c \\ hc \in \mathsf{HeadCmd} & ::= & \ell\texttt{=}v \mid \ell\texttt{=}op(\bar{v}) \mid \ell\texttt{=}v.f \mid v.f\texttt{=}v \mid \ell\texttt{=new } C\texttt{<}\bar{\pi}\texttt{>} \mid \\ & & \ell\texttt{=}v.m\texttt{(}\bar{v}\texttt{)} \mid \texttt{if (}v\texttt{)\{}c\texttt{\}else\{}c\texttt{\}} \end{array}$$

The meaning of the Java commands is exactly as in Java. To simplify later developments, our grammar for writing programs imposes that (1) every intermediate result is assigned to a local variable and (2) the right hand sides of assignments contain no read-write variables.

Class `2DPoint` below shows a program written in our language (where specifications are omitted):

```
class 2DPoint{
  int x; int y;
  void set(int xp, int yp){ this.x = xp; this.y = yp; null }
  bool isOrigin(){
   int ℓ; ℓ = this.x;
   int ℓ′; ℓ′ = this.y;
   int ι = ℓ; int ι′ = ℓ′;
   bool ℓ″ = (ι == 0 & ι′ == 0);
   bool ι″ = ℓ″;
   ι″
  }
}
```

Figure 2.1: Class `2DPoint` written in our language

For clarity reasons, in the remainder of this thesis, in snapshots of code, we omit the terminating `null` value for methods with return type `void`, we omit dereferences to `this` (as real Java permits), and we do not distinguish between read-only and read-write variables. Au contraire, in formal material, we stick to the syntax defined in this section.

2.2 Semantics

This section describes the small-step operational semantics of our Java-like language.

Runtime structures. We model dynamics by a small-step operational semantics that operates on states, consisting of a heap, a command, and a stack. This suffices because we only consider sequential programs. In Chapter 4 and Chapter 5, we will consider multithreaded programs and extend states. As usual, *heaps* map each object identifier to its dynamic type and to a mapping from fields to closed values (i.e., values without variables):

$$h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup \mathsf{ClVal}) \qquad \mathsf{ClVal} = \mathsf{Val} \setminus \mathsf{RdVar}$$

Given a heap h and an object identifier o, we write $h(o)_1$ to denote o's dynamic type and $h(o)_2$ to denote o's store.

Stacks map read/write variables to closed values. Their domains do not include read-only variables, because our operational semantics instantiates those by substitution:

$$s \in \mathsf{Stack} = \mathsf{RdWrVar} \rightharpoonup \mathsf{ClVal}$$

Finally, a *state* consists of a heap, a command, and a stack:

$$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{Cmd} \times \mathsf{Stack}$$

Auxiliary syntax for method call/return. To model method call and return, traditional operational semantics for object-oriented programs use pile of stacks. The purpose of these stacks is to keep track of the current receiver, method parameters, and local variables. Here, we only keep track of the receiver and avoid keeping track of method parameters and local variables. Because method parameters are read-only, we can deal with them with substitution (see reduction rule (Red Call)); while we deal with read-write local variables by (1) ensuring that all read-write local variables have different names (see reduction rule (Red Dcl)) and (2) identifying programs up to variable renaming. This avoids introducing another runtime structure and avoids representing method calls by a derived form that "flattens" the stack. This derived form's purpose is solely to keep track of when the receiver changes. The derived form is defined as follows[1]:

$c ::= \ldots \mid \ell = \texttt{return}(v); c \mid \ldots$
Restriction: This clause must not occur in source programs.

Now we define a derived form, $\ell \leftarrow c; c'$, which assigns the result of a computation c to variable ℓ. In our applications of this derived form, c is always a source program command

[1] Where dots ... indicate that we extend a definition that was given previously. We use this notation all along the thesis.

and we can therefore assume that c does not contain `return`-commands. Below, we write $\mathsf{fv}(c)$ for the the set of free variables of c.

$$\begin{array}{rcll} \ell \leftarrow v;\, c & \triangleq & \ell = \texttt{return}(v);\, c & \\ \ell \leftarrow (T\ \ell';\, c);\, c' & \triangleq & T\ \ell';\, \ell \leftarrow c;\, c' & \text{if } \ell' \notin \mathsf{fv}(c') \text{ and } \ell' \neq \ell \\ \ell \leftarrow (T\ \imath = \ell';\, c);\, c' & \triangleq & T\ \imath = \ell';\, \ell \leftarrow c;\, c' & \text{if } \imath \notin \mathsf{fv}(c') \\ \ell \leftarrow (hc;\, c);\, c' & \triangleq & hc;\, \ell \leftarrow c;\, c' & \end{array}$$

We define sequential composition of commands:

$$c;\, c' \quad \triangleq \quad \texttt{void}\ \ell;\, \ell \leftarrow c;\, c' \quad \text{where } \ell \notin \mathsf{fv}(c, c')$$

Initialization. We define functions to initialize objects and to initialize programs.

Function $\mathsf{df} : \mathsf{Type} \to \mathsf{ClVal}$ maps types to their default values:

$$\mathsf{df}(C\texttt{<}\pi\texttt{>}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{void}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{int}) \triangleq 0 \quad \mathsf{df}(\texttt{bool}) \triangleq \texttt{false}$$

Function $\mathsf{initStore} : \mathsf{Type} \rightharpoonup \mathsf{ObjStore}$ maps object types to their initial object stores. Function $\mathsf{initStore}$ uses function fld (formally defined in Appendix A) to look up fields in the class table.

$$\mathsf{initStore}(t\texttt{<}\bar{\pi}\texttt{>})(f) = \mathsf{df}(T) \text{ iff } (T\ f) \in \mathsf{fld}(C\texttt{<}\pi\texttt{>})$$

Function $\mathsf{init} : \mathsf{Cmd} \to \mathsf{State}$ maps programs to their initial state. Initially, the heap is empty (hence the first $\emptyset$) and the stack is empty (hence the second $\emptyset$):

$$\mathsf{init}(c) = \langle \emptyset, c, \emptyset \rangle$$

Semantics of values. The semantics of read-only variables is left undefined, because we deal with these variables by substitution. For values that are not read-only variables, their semantics is simply identity:

Semantics of Values, $[\![v]\!] \in \mathsf{ClVal}$:

$$[\![\texttt{null}]\!] \triangleq \texttt{null} \quad [\![n]\!] \triangleq n \quad [\![b]\!] \triangleq b \quad [\![o]\!] \triangleq o$$

Semantics of operators. To define the semantics of the command assigning the result of an operation (case $\ell = op(\bar{v})$ of our command language), we define the semantics of operators.

Let arity be a function that assigns to each operator its arity. We define:

$$\begin{array}{lll} \mathsf{arity}(\texttt{==}) \triangleq 2 & \mathsf{arity}(\texttt{\&}) \triangleq 2 & \mathsf{arity}(\texttt{|}) \triangleq 2 \\ \mathsf{arity}(\texttt{!}) \triangleq 1 & \mathsf{arity}(C\ \texttt{classof}) \triangleq 1 & \mathsf{arity}(\texttt{instanceof}\ T) \triangleq 1 \end{array}$$

Let type be a function that maps each operator op to a partial function $\mathsf{type}(op)$ of type $\{\texttt{int}, \texttt{bool}, \texttt{Object}, \texttt{perm}\}^{\mathsf{arity}(op)} \rightharpoonup \{\texttt{int}, \texttt{bool}, \texttt{perm}\}$. We define:

$$\begin{array}{ll} \mathsf{type}(\texttt{==}) \triangleq \{\ ((T, T), \texttt{bool}) \mid T \in \{\texttt{int}, \texttt{bool}, \texttt{Object}, \texttt{perm}, \texttt{lockset}\}\ \} & \\ \mathsf{type}(\texttt{!}) \triangleq \{\ (\texttt{bool}, \texttt{bool})\ \} & \mathsf{type}(\texttt{\&}) \triangleq \mathsf{type}(\texttt{|}) \triangleq \{\ ((\texttt{bool}, \texttt{bool}), \texttt{bool})\ \} \\ \mathsf{type}(C\ \texttt{classof}) \triangleq \{\ (\texttt{Object}, \texttt{bool})\ \} & \mathsf{type}(\texttt{instanceof}\ T) \triangleq \{\ (\texttt{Object}, \texttt{bool})\ \} \end{array}$$

We assume that each operator op is interpreted by a function of the following type:

$$[\![op]\!] \in \mathsf{Heap} \to \bigcup_{(\bar{T},U) \in \mathsf{type}(op)} [\![\bar{T}]\!]^h \to [\![U]\!]^h$$

For the logical operators `!`, `|` and `&`, we assume the usual interpretations. Operator `==` is interpreted as the identity relation. The semantics of `isclassof` and `instanceof` is as follows:

$$[\![C\ \texttt{classof}]\!]^h(o) \triangleq \begin{cases} \texttt{true} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \text{ for some } \bar{\pi} \\ \texttt{false} & \text{if } o \neq \texttt{null},\ h(o)_1 = D\texttt{<}\bar{\pi}\texttt{>}, \text{ and } D \neq C \\ \texttt{false} & \text{if } o = \texttt{null} \\ \mathsf{undef} & o \notin \mathsf{dom}(h) \end{cases}$$

$$[\![o\ \texttt{instanceof}\ ]\!]^h(T) \triangleq \begin{cases} \texttt{true} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 <: T \\ \texttt{false} & \text{if } o \neq \texttt{null} \text{ and } h(o)_1 \not<: T \\ \texttt{false} & \text{if } o = \texttt{null} \\ \mathsf{undef} & \text{if } o \notin \mathsf{dom}(h) \end{cases}$$

We require that the semantics of operators is closed under heap extensions, and that it does not depend on values in the heap:

(a) If $[\![op]\!]^h(\bar{v}) = w$ and $h \subseteq h'$, then $[\![op]\!]^{h'}(\bar{v}) = w$.
(b) If $h' = h[o.f \mapsto u]$, then $[\![op]\!]^h = [\![op]\!]^{h'}$.

The interpretations of "C `classof`" and "`instanceof` T" depend on the heap, but still satisfy requirement (b), as they only depend on the type components of objects which is fixed at object creation. Formally, the semantics of operators is expressed as follows:

Semantics of Operators: $[\![op(\bar{v})]\!] : \mathsf{Heap} \to \mathsf{Stack} \rightharpoonup \mathsf{ClVal}$**:**

(Sem Op)
$$\frac{[\![w_1]\!]^h_s = v_1 \quad \cdots \quad [\![w_n]\!]^h_s = v_n \quad [\![op]\!]^h(v_1, \ldots, v_n) = v}{[\![op(w_1, \ldots, w_n)]\!]^h_s = v}$$

Small-step reduction. The state reduction relation $\to_{ct}$ is given with respect to a class table ct. We usually omit the subscript ct. In the reduction rules, we use the following abbreviation for field updates: $h[o.f \mapsto v] = h[o \mapsto (h(o)_1, h(o)_2[f \mapsto v])]$.

Below, we use function $\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>})$ (formally defined in Appendix A) to look up the code of method m in class $C\texttt{<}\bar{\pi}\texttt{>}$.

State Reductions, $st \to_{ct} st'$**:**

(Red Dcl) $\ell \notin \mathsf{dom}(s) \quad s' = s[\ell \mapsto \mathsf{df}(T)]$
$\langle h,\ T\ \ell;\ c, s\rangle \to \langle h,\ c, s'\rangle$

(Red Fin Dcl) $s(\ell) = v \quad c' = c[v/\imath]$
$\langle h,\ T\ \imath = \ell;\ c, s\rangle \to \langle h,\ c', s\rangle$

(Red Var Set) $s' = s[\ell \mapsto v]$
$\langle h,\ \ell = v;\ c, s\rangle \to \langle h,\ c, s'\rangle$

(Red Op) $\mathsf{arity}(op) = |\bar{v}| \quad [\![op]\!]^h(\bar{v}) = w \quad s' = s[\ell \mapsto w]$
$\langle h,\ \ell = op(\bar{v});\ c, s\rangle \ \rightarrow\ \langle h,\ c, s'\rangle$

(Red Get) $s' = s[\ell \mapsto h(o)_2(f)]$
$\langle h,\ \ell = o.f;\ c, s\rangle \ \rightarrow\ \langle h,\ c, s'\rangle$

(Red Set) $h' = h[o.f \mapsto v]$
$\langle h,\ o.f = v;\ c, s\rangle \ \rightarrow\ \langle h',\ c, s\rangle$

(Red New) $o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o]$
$\langle h,\ \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c, s\rangle \ \rightarrow\ \langle h',\ c, s'\rangle$

(Red Call) $h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath_0; \bar{\imath}).c_m \quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}]$
$\langle h,\ \ell = o.m(\bar{v});\ c, s\rangle \ \rightarrow\ \langle h,\ \ell \leftarrow c';\ c, s\rangle$

(Red Return)
$\langle h,\ \ell = \texttt{return}(v);\ c, s\rangle \ \rightarrow\ \langle h,\ \ell = v;\ c, s\rangle$

(Red If True)
$\langle h,\ \texttt{if (true)\{}c\texttt{\}else\{}c'\texttt{\}};\ c'', s\rangle \ \rightarrow\ \langle h,\ c;\ c'', s\rangle$

(Red If False)
$\langle h,\ \texttt{if (false)\{}c\texttt{\}else\{}c'\texttt{\}};\ c'', s\rangle \ \rightarrow\ \langle h,\ c';\ c'', s\rangle$

Remarks:

- In (Red Dcl), read-write variables are initialized to a default value.
- In (Red Fin Dcl), declaration of read-only variables is handled by substituting the right-hand side's value for the newly declared variable in the continuation.
- In (Red New), the heap is extended to contain a new object.
- In (Red Call), $\imath_0$ is the formal method receiver and $\bar{\imath}$ are the formal method parameters. Like for declaration of read-only variables, both the formal method receiver and the formal method parameters are substituted by the actual receiver and the actual method parameters.
- We do not have loops and do not use them in later examples. We note, however, that they could be added straightforwardly.

2.3 Related Work and Conclusion

Related Work. A number of formalization of Java exists. Here, we give a quick comparison with some of these formalizations that are related to ours.

The closely related work is Parkinson's thesis [88] who formalizes a subset of Java to specify and verify such programs with separation logic. There are, however, a few differences: we feature value-parameterized classes, we do not include casts (but it would be straightforward to add them, as we did in our technical report [54]), we do not model constructors, we do not provide block scoping, and, contrary to Parkinson, programs written in our model language are not valid Java programs. In the next chapters, we will compare our work with Parkinson's work again.

Model languages to study Java's type system include Featherweight Java [66] and Drossopoulou et al.'s work [45]. These works focus on type soundness, whereas we focus on providing a formal language amenable to program verification.

A number of formalization of Java inside theorem provers exists. Huisman formalizes a large subset of Java inside Isabelle/HOL and PVS [60]. She mechanically proves soundness

of Hoare rules for her model language. Klein and Nipkow formalize a large subset of Java in Isabelle/HOL called Jinja [73]. Their formalization includes a virtual machine and a compiler, and they prove type soundness of all the components.

Conclusion. We presented a Java-like language that will serve at the basis for Chapters 3, 4, 5, 6, and 7. This language contains the core object-oriented features of Java: mutable fields, inheritance and method overriding, and interfaces. For expressivity reasons, it features value-parameterized classes. In the next chapter, we show how to specify and verify programs written in this language with separation logic.

Chapter 3

Separation Logic for a Java-like Language

In this chapter, we show how we specify and verify programs written in the Java-like language from Chapter 2 with separation logic. This chapter will serve as the basis for Chapters 4, 5, 6, and 7.

This chapter is structured as follows: in Section 3.1 we informally present separation logic and in Section 3.2 we formally present the specific flavor of separation logic we use to specify programs. In Section 3.3 we show Hoare rules that serve to verify programs, in Section 3.4 and 3.5 we describe what are verified classes and verified programs. In Section 3.6, we present two examples of verified classes: a simple `Roster` example and an advanced `Iterator` example. Finally, we discuss related work and conclude in Section 3.7.

Note: The work described in this chapter has been done in collaboration with Christian Haack. It was published in the International Conference on Algebraic Methodology and Software Technology (AMAST 2008) [53]. A considerably extended version of the `Iterator` example from Section 3.6.2 has been published in the Journal of Object Technology (JOT) [55].

Convention: In formal material, we use dots "..." to indicate that definitions from the previous chapter are being extended.

3.1 Separation Logic: Informally

In this section, we informally present what is separation logic and what makes it a convenient tool to reason about object-oriented multithreaded programs. In particular, we show aspects of separation logic that we use in this thesis.

3.1.1 Separation Logic — Formulas as Access Tickets

Separation logic [93] combines the usual logical operators with the points-to predicate $x.f \mapsto v$ and the resource conjunction $F * G$.

The predicate $x.f \mapsto v$ has a *dual purpose*: firstly, it asserts that the object field $x.f$ contains data value v and, secondly, it represents a *ticket* that grants permission to access

the field $x.f$. This is formalized by separation logic's Hoare rules for reading and writing fields (where $x.f \mapsto \_$ is short for $(\exists v)(x.f \mapsto v)$):

$$\{x.f \mapsto \_\}x.f = v\{x.f \mapsto v\} \qquad \{x.f \mapsto v\}y = x.f\{x.f \mapsto v \ * \ v == y\}$$

The crucial difference to standard Hoare logic is that both these rules have a precondition of the form $x.f \mapsto \_$: this formula functions as an *access ticket* for $x.f$.

It is important that tickets are not forgeable: one ticket is not the same as two tickets! For this reason, the resource conjunction $*$ is not idempotent: F is not equivalent to $F * F$. Intuitively, the formula $F * G$ represents two access tickets F and G to *separate* parts of the heap. In other words, the part of the heap that F permits to access is *disjoint* from the part of the heap that G permits to access. As a consequence, separation logic's $*$ allows reasoning about (the absence of) aliases easily: this is why the Hoare rules shown above are sound. To further exemplify $*$'s importance, consider the following code (where we write specifications in dark blue):

```
class Amount{
  int x;
  requires ?;
  ensures  a.x contains 0 and b.x contains 1;
  void m(Amount a,Amount b){
   a.x = 0;
   b.x = 1;
  }
}
```

Now, imagine that we would like to prove that, after executing `m`, `a.x` contains `0` and `b.x` contains `1` (as informally specified by `m`'s `ensures` clause). To prove such a postcondition, we have to make sure that `a` and `b` point to different objects at entry to `m`. This can be specified by $\texttt{a.x} \mapsto \_ * \texttt{b.x} \mapsto \_$ as `m`'s precondition. Note, however, that $\texttt{a.x} \mapsto \_ \wedge \texttt{b.x} \mapsto \_$ would be incorrect as precondition, because `a` and `b` could point to the same object, so that `a.x` could contain `1` after `m`'s execution.

3.1.2 Separation Logic — Local Reasoning

Another crucial property of separation logic is that it allows to reason locally about methods. This means that, when calling a subprogram, one can identify (1) the (small) part of the heap accessed by that subprogram and (2) the rest of the heap that is left unaffected. Formally, this is expressed by the (Frame) rule [93] below:

$$\frac{\{F\}c\{F'\}}{\{F * G\}c\{F' * G\}} \text{ (Frame)}$$

This rule expresses that given a command c which only accesses the part of the heap described by F, one can reason locally about command c ((Frame)'s premise) and deduce something global, i.e., in the context of a bigger heap $F * G$ ((Frame)'s conclusion). In this rule, G is called the frame and represents the part of the heap unaffected by executing c. Later, it will be important to show that the (Frame) rule can be added to our verification rules without harming soundness.

3.1.3 Separation Logic — Abstraction

A good object-oriented programming practice is to consider objects abstractly i.e., to hide implementation details to clients. To this end, Parkinson introduced *abstract predicates* [88]. Abstract predicates hide implementation details to clients but allow class implementers to use them. In other words, abstract predicates are *opaque* to clients but *transparent* to class implementers.

As an example, consider class `Box` below that defines a predicate `state` to hide its concrete state.

```
class Box{
  int blx; int bly; // bottom left point
  int trx; int try; // top right point
  pred state = this.blx ↦ _ * this.bly ↦ _ * this.trx ↦ _ * this.try ↦ _;
  requires this.state;
  ensures  this.state;
  void move(deltax, deltay){
   this.blx += deltax; this.trx += deltax;
   this.bly += deltay; this.try += deltay;
  }
}
```

Predicate `state` hides that the internal state of a `Box` consists of the four fields `blx`, `bly`, `trx` and `try`: by inspecting method `move`'s contract (i.e., its `requires` and `ensures` clauses), one cannot tell what the internal representation of class `Box` looks like. In particular, class `Box` could store the coordinates of other points than the bottom left point and the top right point or it could store one point and two distances.

3.1.4 Separation Logic — Proofs Outlines

Since separation logic's early days [93], proofs of programs are represented as *proof outlines*. In proof outlines, assertions (or formulas) are inserted in-between commands. Such assertions symbolically represent the heap. In a nutshell, commands can be seen as partial functions from symbolic heaps to symbolic heaps: a command takes a symbolic heap as input and – if the symbolic heap matches the command's precondition – returns a symbolic heap as output. All along this thesis, we represent proof of programs with proof outlines. As an example, consider some method `m` with precondition F, body $c_0; \ldots; c_n$ and postcondition G. A proof outline for `m` will look like this:

```
 { F }
c0;
 { ... }
...
 { ... }
cn;
 { H } ← Does H entails G ?
```

The last line indicates that, for a proof outline to be correct, the last output formula (H) must entail the method's postcondition (G).

3.2 Separation Logic

3.2.1 Syntax

In this section, we present the flavor of separation logic that we use to write method contracts. We use *intuitionistic* separation logic [67, 93, 88] as this is suitable to reason about properties that are invariant under heap extensions, and is appropriate for garbage-collected languages like Java. Contrary to classical separation logic, intuitionistic separation logic admits weakening. Informally, this means that one can "forget" a part of the state: this is why intuitionistic separation logic is appropriate for garbage-collected languages. The semantical difference between intuitionistic and classical separation logic is detailed in Section 3.2.5.

Expressions e are built from values and variables using arithmetic and logical operators:

$$e \in \mathsf{Exp} \ ::= \ \pi \ \mid \ \ell \ \mid \ op(\bar{e})$$

Specification formulas F are defined by the following grammar:

$$lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{|}\} \qquad qt \in \{\texttt{ex}, \texttt{fa}\} \qquad \kappa \in \mathsf{Pred} \ ::= \ P \mid P\texttt{@}C$$
$$F \in \mathsf{Formula} \ ::= \ e \ \mid \ \texttt{PointsTo(}e.f, \pi, e\texttt{)} \ \mid \ \pi.\kappa\texttt{<}\bar{\pi}\texttt{>} \ \mid \ F \ lop \ F \ \mid \ \texttt{(}qt \ T \ \alpha\texttt{)(}F\texttt{)}$$

We now explain these formulas:

The *points-to predicate* $\texttt{PointsTo(}e.f, \pi, v\texttt{)}$ is ASCII for $e.f \overset{\pi}{\longmapsto} v$ [21]. Superscript π must be a fractional permission [23] i.e., a fraction $\frac{1}{2^n}$ in the interval $(0, 1]$. As explained earlier, formula $\texttt{PointsTo(}e.f, \pi, v\texttt{)}$ has a dual meaning: firstly, it asserts that field $e.f$ contains value v, and, secondly, it represents access right π to $e.f$. Permission $\pi = 1$ grants write access while any permission π grants read access.

The *resource conjunction* $F \texttt{*} G$ (a.k.a *separating conjunction*) expresses that resources F and G are independently available: using either of these resources leaves the other one intact. Resource conjunction is not idempotent: F does *not* imply $F \texttt{*} F$. Because Java is a garbage-collected language, we allow dropping assertions: $F \texttt{*} G$ implies F.

The *resource implication* $F \texttt{-*} G$ (a.k.a. *linear implication* or *magic wand*) means "consume F yielding G". Resource $F \texttt{-*} G$ permits to trade resource F to receive resource G in return. Resource conjunction and implication are related by the modus ponens: $F \texttt{*} \texttt{(}F \texttt{-*} G\texttt{)}$ implies G. Section 3.6.2 exemplifies contracts that make heavy use of the magic wand.

We remark that, to avoid a proof theory with bunched contexts (see Section 3.2.6), we omit the $\Rightarrow$-implication between heap formulas (and did not need it in later examples). However, this design decision is not essential.

The *predicate application* $\pi.\kappa\texttt{<}\bar{\pi}\texttt{>}$ applies abstract predicate κ to its receiver parameter π and the additional parameters $\bar{\pi}$. As explained above, predicate definitions in classes map abstract predicates to concrete definitions. Predicate definitions can be extended in subclasses to account for extended object state. Semantically, P's predicate extension in class C gets $\texttt{*}$-conjoined with P's predicate extensions in C's superclasses. The *qualified predicate* $\pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$ represents the $\texttt{*}$-conjunction of P's predicate extensions in C's superclasses, up to and including C. The *unqualified predicate* $\pi.P\texttt{<}\bar{\pi}\texttt{>}$ is equivalent to $\pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$, where C is π's dynamic class. We allow predicates with missing parameters: Semantically, missing parameters are existentially quantified.

For expressivity purposes, it is crucial to allow parameterization of objects by permissions. For this, we include a special type `perm` for fractional permissions:

$$T, U, V, W \in \mathsf{Type} \quad ::= \quad \ldots \mid \texttt{perm} \mid \ldots$$

Because generic parameters are instantiated by specification values (see Section 2.1), we extend specification values with fractional permissions. Fractional permissions are represented symbolically: 1 represents itself, and if symbolic fraction π represents concrete fraction fr then `split`(π) represents $\frac{1}{2} \cdot fr$.

$$\pi \in \mathsf{SpecVal} \quad ::= \quad \ldots \mid 1 \mid \texttt{split}(\pi) \mid \ldots$$

Quantifiers `(`$qt\ T\ \alpha$`)(`F`)` are standard. Because specification values π and expressions e contain logical variables α (see pages 10 and 20), bound variables can appear in many positions: as type parameters; as the first, second, and third parameter in `PointsTo` predicates; as predicate parameters etc.

In our model language, specifications are expressed with methods pre and postconditions. Preconditions are declared with keyword `requires` and postconditions are declared with keyword `ensures`. In postconditions, the special identifier `result` can be used to refer to the value returned. As shown in Section 2.1, the syntax of methods with pre and postconditions is as follows:

$$\texttt{class}\ C\ \texttt{\{}\ \ldots\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>requires}\ F\texttt{;ensures}\ F\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{v}\texttt{)\{}c\texttt{\}}\ \ldots\ \texttt{\}}$$

Interfaces may declare *abstract predicates* and classes may implement them by providing concrete definitions as separation logic formulas.

$$\texttt{interface I \{ ... pred }P\texttt{<}\bar{T}\ \bar{x}\texttt{>; ... \}}$$
$$\texttt{class C implements I \{ ... pred }P\texttt{<}\bar{T}\ \bar{x}\texttt{>=}F\texttt{; ... \}}$$

Class axioms export facts about relations between abstract predicates, without revealing the detailed predicate implementations. Class implementors have to prove class axioms and class clients can use them.

$$\texttt{class C \{ ... axiom }F\texttt{;... \}}$$

Predicate definitions can be preceded by an optional `public` modifier. The role of the `public` modifier is to export the definition of a predicate *in a given class* to clients. Formally, `public` desugars to an `axiom` (where C is the enclosing class):

$$\texttt{public pred }P\texttt{<}\bar{T}\ \bar{x}\texttt{>=}F \quad \triangleq \quad \begin{array}{l}\texttt{pred }P\texttt{<}\bar{T}\ \bar{x}\texttt{>=}F\texttt{;}\\ \texttt{axiom }P\texttt{@}C\texttt{<}\bar{x}\texttt{>}\ \texttt{*-*}\ F\end{array}$$

Although `public` desugars to an axiom, `public pred P<`$\bar{T}\ \bar{x}$`> =` F`;` is not equivalent to `pred P<`$\bar{T}\ \bar{x}$`> =` F`;` and `axiom P<`$\bar{T}\ \bar{x}$`> *-*` F`;`. The latter axiom is more general than `public`'s desugaring because it constrains extensions of `P` in C's subclasses to satisfy this axiom (whereas `public`'s desugaring simply exports `P`'s definition *in class* C). In Sections 3.6.1 and 4.6.2, we make heavy use of this distinction.

Now, we show how to specify a class using our language.

For this, we reuse the `Box` example shown in Section 3.1.3. In Section 3.1.3, method `move`'s contract is incomplete: it just specifies that fields of class `Box` are accessed but it

does not specify method `move`'s functional behavior (i.e., how it increases x-coordinates by `deltax` and y-coordinates by `deltay`). Expressing method `move`'s behavior can be done as follows:

```
class Box{
  int blx; int bly; // bottom left point
  int trx; int try; // top right point
  pred state<int i, int j, int k, int l> =
            PointsTo(this.blx, 1, i) * PointsTo(this.bly, 1, j) *
            PointsTo(this.trx, 1, k) * PointsTo(this.try, 1, l);
  requires this.state<i,j,k,l>;
  ensures  (ex int i',j',k',l')(this.state<i',j',k',l'> &
                               i+deltax==i' & j+deltax==j' &
                               k+deltax==k' & l+deltax==l');
  void move(deltax, deltay){
   blx += deltax; trx += deltax;
   bly += deltay; try += deltay;
  }
}
```

The previous example shows a basic usage of parameterized predicates. Advanced specifications use predicate parameters in combination with class parameters to accommodate different usages of class instances. For example, for class `Box` to express that boxes may be movable or sticky, one can parameterize class `Box` with a permission parameter:

```
class Box<perm p>{
  int blx; int bly; // bottom left point
  int trx; int try; // top right point
  pred state<perm q> = PointsTo(this.blx, q, _) * PointsTo(this.bly, q, _) *
                       PointsTo(this.trx, q, _) * PointsTo(this.try, q, _);
  requires init;
  ensures  this.state<p>;
  init(int i, int j, int k, int l){ blx = i; bly = j; trx = k; try = l; }
  requires this.state<q>;
  ensures  this.state<q>;
  void display(){ ... }
  requires this.state<1>;
  ensures  this.state<1>;
  void move(deltax, deltay){
   blx += deltax; trx += deltax;
   bly += deltay; try += deltay;
  }
}
```

Now, the programmer can formally distinguish movable boxes (by instantiating `p` with 1) from sticky boxes (by instantiating `p` with a fraction strictly less than 1). Because `move`'s precondition requires `this.state<1>`, it cannot be called on a sticky box: `Box`'s

constructor (i.e., method `init`, because our model language does not have constructors) ensures that, for sticky boxes, `state<1>` can never be established. This formally enforces that sticky boxes cannot be made movable. In method `init`'s precondition, *predicate* `init` represents write access to all fields of class `Box`[1].

We define convenient *derived forms* for specification formulas:

$$\texttt{PointsTo}(e.f, \pi, T) \triangleq \texttt{(ex}\ T\ \alpha\texttt{)(}\ \texttt{PointsTo}(e.f, \pi, \alpha)\texttt{)}$$
$$\texttt{Perm}(e.f, \pi) \triangleq \texttt{(ex}\ T\ \alpha\texttt{)(}\ \texttt{PointsTo}(e.f, \pi, \alpha)\texttt{)} \quad \text{where } T \text{ is } e.f\text{'s least type}$$
$$F\ \texttt{*-*}\ G \triangleq \texttt{(}F\ \texttt{-*}\ G\texttt{)\&(}G\ \texttt{-*}\ F\texttt{)}$$
$$F\ \texttt{assures}\ G \triangleq F\ \texttt{-*}\ \texttt{(}F\ \texttt{*}\ G\texttt{)} \qquad F\ \texttt{ispartof}\ G \triangleq G\ \texttt{-*}\ \texttt{(}F\ \texttt{*}\ \texttt{(}F\ \texttt{-*}\ G\texttt{))}$$

Intuitively, F `ispartof` G says that F is a physical part of G: one can take G apart into F and its complement F `-*` G, and can put the two parts together to obtain G back.

3.2.2 Types

Because the semantics of formulas depends on a typing judgment, we need to define typing rules before giving the formulas's semantics.

A *type environment* is a partial function of type $\mathsf{ObjId} \cup \mathsf{Var} \rightharpoonup \mathsf{Type}$. We use the meta-variable Γ to range over type environments. Γ_{hp} denotes the *restriction of* Γ *to* ObjId:

$$\Gamma_{\mathsf{hp}} \triangleq \{\ (o, T) \in \Gamma \mid o \in \mathsf{ObjId}\ \}$$

A type environment is *good* when objects within its domain are well-typed:

Good Environments, $\Gamma \vdash \diamond$:

(Env)
$$\frac{(\forall x \in \mathsf{dom}(\Gamma))(\Gamma \vdash \Gamma(x) : \diamond) \quad (\forall o \in \mathsf{dom}(\Gamma))(\Gamma(o) <: \texttt{Object} \text{ and } \Gamma_{\mathsf{hp}} \vdash \Gamma(o) : \diamond)}{\Gamma \vdash \diamond}$$

We define a sanity condition on types: primitive types are always sane, while user-defined types must be such that (1) type identifiers are in the class table and (2) type parameters are well-typed. Below, the existential quantification in (Ty Ref)'s second premise enforces typing derivations to be finite.

Good Types, $\Gamma \vdash T : \diamond$:

(Ty Primitive)
$$\frac{T \in \{\texttt{void}, \texttt{int}, \texttt{bool}, \texttt{perm}\}}{\Gamma \vdash T : \diamond}$$

(Ty Ref)
$$\frac{t\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad (\exists \Gamma' \subset \Gamma)(\Gamma' \vdash \diamond \quad \Gamma' \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])}{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}$$

We define a *heap extension order* on well-formed type environments:

$$\Gamma' \supseteq_{\mathsf{hp}} \Gamma \quad \text{iff} \quad \Gamma' \vdash \diamond,\ \Gamma \vdash \diamond,\ \Gamma' \supseteq \Gamma \text{ and } \Gamma'_{|\mathsf{Var}} = \Gamma_{|\mathsf{Var}}$$

As models of formulas are tuples that contain a heap and a stack (see Section 3.2.3), we define a well-typedness judgment for objects, heaps, and stacks:

[1] This predicate is emitted after creating an object as rule (New) on page 35 will formalize. Details about this predicate are given in Section 3.3.

Well-typed Objects, $\Gamma \vdash obj : \diamond$:

$$\text{(Obj)}\quad \frac{\Gamma \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond \quad (\forall f \in \mathsf{dom}(os))(T\ f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) \ \Rightarrow\ \Gamma \vdash os(f) : T) \quad \mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))}{\Gamma \vdash (C\texttt{<}\bar{\pi}\texttt{>}, os) : \diamond}$$

Note that we require $\mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$, not $\mathsf{dom}(os) = \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$. Thus, we allow partial objects. This is needed, because $*$ joins heaps on a per-field basis. This will be needed for fine-grained concurrency (as Section 3.6.1 exemplifies).

Below, we use function $\mathsf{fst} : \mathsf{Heap} \to (\mathsf{ObjId} \rightharpoonup \mathsf{Type})$ to extract the function that maps object identifiers to their dynamic types from a heap:

$$h(o) = (T, \_) \Rightarrow \mathsf{fst}(h)(o) = T$$

We now define well-typed heaps and stacks:

Well-typed Heaps and Stacks, $\Gamma \vdash h : \diamond$ and $\Gamma \vdash s : \diamond$:

$$\text{(Heap)}\quad \frac{\Gamma \vdash \diamond \quad \Gamma \subseteq \mathsf{fst}(h) \quad (\forall o \in \mathsf{dom}(h))(\Gamma \vdash h(o) : \diamond)}{\Gamma \vdash h : \diamond}$$

$$\text{(Stack)}\quad \frac{\Gamma \vdash \diamond \quad (\forall x \in \mathsf{dom}(s))(\Gamma \vdash s(x) : \Gamma(x))}{\Gamma \vdash s : \diamond}$$

Because formulas include expressions, we define a well-typedness judgment for values, specification values, and expressions (recall that expressions include specification values of type `bool`).

Well-typed Values and Specification Values, $\Gamma \vdash v : T$ and $\Gamma \vdash \pi : T$:

$$\text{(Val Var)}\quad \frac{\Gamma \vdash \diamond \quad \Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(Val Oid)}\quad \frac{\Gamma \vdash \diamond \quad \Gamma(o) = T}{\Gamma \vdash o : T} \qquad \text{(Val Sub)}\quad \frac{\Gamma \vdash \pi : T \quad T <: U}{\Gamma \vdash \pi : U} \qquad \text{(Val Null)}\quad \frac{\Gamma \vdash t\texttt{<}\bar{\pi}\texttt{>} : \diamond}{\Gamma \vdash \texttt{null} : t\texttt{<}\bar{\pi}\texttt{>}}$$

$$\text{(Val Int)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \texttt{int}} \qquad \text{(Val Bool)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash b : \texttt{bool}} \qquad \text{(Val Full)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash 1 : \texttt{perm}} \qquad \text{(Val Split)}\quad \frac{\Gamma \vdash \pi : \texttt{perm}}{\Gamma \vdash \texttt{split}(\pi) : \texttt{perm}}$$

Well-typed Expressions, $\Gamma \vdash e : T$:

$$\text{(Exp Sub)}\quad \frac{\Gamma \vdash e : T \quad T <: U}{\Gamma \vdash e : U} \qquad \text{(Exp Var)}\quad \frac{\Gamma \vdash \diamond \quad \Gamma(\ell) = T}{\Gamma \vdash \ell : T} \qquad \text{(Exp Op)}\quad \frac{\Gamma \vdash \bar{e} : \bar{U} \quad \mathsf{type}(op)(\bar{U}) = T}{\Gamma \vdash op(\bar{e}) : T}$$

We now have all the machinery to define well-typed formulas. Below, the partial function $\mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>})$ (formally defined in Appendix A) looks up the type of predicate P in the least supertype of $C\texttt{<}\bar{\pi}\texttt{>}$ that defines or extends P.

Well-typed Formulas, $\Gamma \vdash F : \diamond$:

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash e : \diamond}\ (\text{Form Bool}) \qquad \frac{\Gamma \vdash e : U \quad \Gamma \vdash \pi : \texttt{perm} \quad T\ f \in \mathsf{fld}(U) \quad \Gamma \vdash e' : T}{\Gamma \vdash \texttt{PointsTo}(e.f, \pi, e') : \diamond}\ (\text{Form Points To})$$

$$\frac{\Gamma \vdash F, F' : \diamond}{\Gamma \vdash F\ lop\ F' : \diamond}\ (\text{Form Log Op}) \qquad \frac{\Gamma \vdash \pi : U \quad \mathsf{ptype}(\kappa, U) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \quad \Gamma \vdash \bar{\pi}' : \bar{T}}{\Gamma \vdash \pi.\kappa\texttt{<}\bar{\pi}'\texttt{>} : \diamond}\ (\text{Form Pred})$$

$$\frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T \vdash F : \diamond}{\Gamma \vdash (qt\ T\ \alpha)(F) : \diamond}\ (\text{Form Quant})$$

After all these boring definitions, we exemplify how to type check formula $o.\texttt{state<}\frac{1}{2}\texttt{>} * \texttt{PointsTo}(o.\texttt{blx}, \frac{1}{2}, 1)$ in a type environment $\Gamma = o \mapsto \texttt{Box<1>}$ (where we reuse Section 3.2.1's parameterized class `Box`). Below, we consider that the class table only consists of class `Box` parameterized by permission `q`. Vertical dots indicate elided parts of derivations.

$$\frac{\Gamma \vdash o.\texttt{state<}\frac{1}{2}\texttt{>} \qquad \Gamma \vdash \texttt{PointsTo}(o.\texttt{blx}, \frac{1}{2}, 1)}{\Gamma \vdash o.\texttt{state<}\frac{1}{2}\texttt{>} * \texttt{PointsTo}(o.\texttt{blx}, \frac{1}{2}, 1)}\ (\text{Form Log Op})$$

We now show that $\Gamma \vdash o.\texttt{state<}\frac{1}{2}\texttt{>}$ and $\Gamma \vdash \texttt{PointsTo}(o.\texttt{blx}, \frac{1}{2}, 1)$ hold:

$$\frac{\dfrac{\Gamma \vdash \diamond \qquad \overline{\Gamma(o) = o : \texttt{Box<1>}}}{\Gamma(o) \vdash o : \texttt{Box<1>}}\ (\text{Val Oid}) \qquad \mathsf{ptype}(\texttt{state}, \texttt{Box<1>}) = \texttt{pred state<perm q>} \qquad \dfrac{\vdots}{\Gamma \vdash \frac{1}{2}, 1 : \texttt{perm}, \texttt{int}}}{\Gamma \vdash o.\texttt{state<}\frac{1}{2}\texttt{>}}$$

$$\frac{\dfrac{\vdots}{\Gamma(o) \vdash o : \texttt{Box<1>}}\ (\text{Val Oid}) \qquad \dfrac{\dfrac{\overline{\Gamma \vdash 1 : \texttt{perm}}\ (\text{Val Full})}{\Gamma \vdash \frac{1}{2} : \texttt{perm}}\ (\text{Val Split}) \qquad \overline{\texttt{int blx} \in \mathsf{fld}(\texttt{Box<1>})}}{\Gamma \vdash \texttt{PointsTo}(o.\texttt{blx}, \frac{1}{2}, 1)}$$

Finally, we show that $\Gamma \vdash \diamond$ holds:

$$\frac{\overline{\texttt{Box<1>} <: \texttt{Object}} \qquad \dfrac{\texttt{Box<perm q>} \in ct \qquad \overline{\emptyset \vdash 1 : \texttt{perm}}\ (\text{Val Full})}{o \mapsto \texttt{Box<1>} \vdash \texttt{Box<1>} : \diamond}\ (\text{Ty Ref})}{o \mapsto \texttt{Box<1>} \vdash \diamond}\ (\text{Env})$$

3.2.3 Resources

In Section 3.2.5, we define a forcing relation of the form $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F$, where $\mathcal{E}$ is a *predicate environment* (that maps predicate identifiers to concrete heap predicates) and $\mathcal{R}$ is a *resource*. In this paragraph, we define *resources* i.e., models of our formulas. Intuitively, if $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F$ holds, this means that resource $\mathcal{R}$ is a program state that is described by F.

Resources $\mathcal{R}$ range over the set $\mathsf{Resource}$ with a binary relation $\# \subseteq \mathsf{Resource} \times \mathsf{Resource}$ (the *compatibility relation*) and a partial binary operator $* : \# \rightharpoonup \mathsf{Resource}$ (the *resource joining operator*) that is associative and commutative. Concretely, resources are pairs $\mathcal{R} = (h, \mathcal{P})$ of a *heap* h and a *permission table* $\mathcal{P} \in \mathsf{ObjId} \times \mathsf{FieldId} \rightharpoonup [0, 1]$. We require that resources satisfy the following axioms:

(a) $\mathcal{P}(o, f) > 0$ if $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$.
(b) For all $o \notin \mathsf{dom}(h)$ and all f, $\mathcal{P}(o, f) = 0$.

Axiom (a) ensures that the (partial) heap h only contains cells that are associated with a positive permission. Technically, this condition is needed to prove soundness of the verification rule for field updates. Axiom (b) ensures that all objects that are not yet allocated have minimal permissions (with respect to the resource order presented below). This is needed to prove soundness of the verification rule for allocating new objects.

Each of the two resource components carries itself a resource structure $(\#, *)$. These structures are lifted to resources component-wise. We now define $\#$ and $*$ for the two components.

Heaps (which, as defined in Section 2.2, have type $\mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup \mathsf{ClVal})$) are compatible if they agree on object types and memory content:

$$h \# h' \text{ iff } \begin{cases} (\forall o \in \mathsf{dom}(h) \cap \mathsf{dom}(h')) \ (\\ h(o)_1 = h'(o)_1 \text{ and } (\forall f \in \mathsf{dom}(h(o)_2) \cap \mathsf{dom}(h'(o)_2))(\ h(o)_2(f) = h'(o)_2(f)\ )\ ) \end{cases}$$

For example, heap $o \mapsto (\texttt{Box}, (\texttt{blx} \mapsto 0))$ and heap $o \mapsto (\texttt{Box}, (\texttt{blx} \mapsto 1))$ are incompatible, because they contain different values in field `blx` of object o. However, heap $o \mapsto (\texttt{Box}, (\texttt{blx} \mapsto 0))$ and heap $o \mapsto (\texttt{Box}, (\texttt{trx} \mapsto 0))$ are compatible, because they contain different values in *different* fields of object o. Note that, in this example, the two heaps each contains a field of the single object o: we allow partial objects.

To define heap joining, we lift set union to deal with undefinedness: $f \vee g = f \cup g$, $f \vee \mathsf{undef} = \mathsf{undef} \vee f = f$. Similarly for types: $T \vee \mathsf{undef} = \mathsf{undef} \vee T = T \vee T = T$.

$$(h * h')(o)_1 \triangleq h(o)_1 \vee h'(o)_1 \qquad (h * h')(o)_2 \triangleq h(o)_2 \vee h'(o)_2$$

Joining *permission tables* is point-wise addition:

$$\mathcal{P} \# \mathcal{P}' \text{ iff } (\forall o)(\mathcal{P}(o) + \mathcal{P}'(o) \leq 1) \qquad (\mathcal{P} * \mathcal{P}')(o) \triangleq \mathcal{P}(o) + \mathcal{P}'(o)$$

For later convenience, we define projection operators $_{\mathsf{hp}}$ and $_{\mathsf{perm}}$ as follows:

$$(h, \mathcal{P})_{\mathsf{hp}} \triangleq h \qquad (h, \mathcal{P})_{\mathsf{perm}} \triangleq \mathcal{P}$$

We define an order on heaps, permission tables, and resources as follows:

$$h \leq h' \triangleq (\exists h'')(h * h'' = h')$$
$$\mathcal{P} \leq \mathcal{P}' \triangleq (\exists \mathcal{P}'')(\mathcal{P} * \mathcal{P}'' = \mathcal{P}') \quad \mathcal{R} \leq \mathcal{R}' \triangleq (\exists \mathcal{R}'')(\mathcal{R} * \mathcal{R}'' = \mathcal{R}')$$

The three order just defined can be understood as follows: a heap h is less than a heap h' if h contains less memory cells than h', a permission table pt is less than a permission table $\mathcal{P}'$ if $\mathcal{P}$'s permissions are less than $\mathcal{P}'$'s permissions, and a resource $\mathcal{R}$ is less than a resource $\mathcal{R}'$ if $\mathcal{R}$'s components are all less than $\mathcal{R}'$'s components.

Example. To give the reader an intuition of what resources are, we give an example. For this, we reuse Figure 2.1's class `2DPoint`:

```
class 2DPoint{
  int x;
  int y;
  ...
}
```

We define three resources and show how these resources are ordered. For this, we define $h, \mathcal{P}, h', \mathcal{P}', h''$, and $\mathcal{P}''$ as follows:

$$h \triangleq o \mapsto (\texttt{Point}, (\texttt{x} \mapsto 0)) \qquad \mathcal{P}(q,f) = \begin{cases} 1/2 & \text{if } q = o \text{ and } f = \texttt{x} \\ 0 & \text{otherwise} \end{cases}$$

$$h' \triangleq o \mapsto (\texttt{Point}, (\texttt{x} \mapsto 0)) \qquad \mathcal{P}'(q,f) = \begin{cases} 1 & \text{if } q = o \text{ and } f = \texttt{x} \\ 0 & \text{otherwise} \end{cases}$$

$$h'' \triangleq o \mapsto (\texttt{Point}, (\texttt{x} \mapsto 0, \texttt{y} \mapsto 0)) \qquad \mathcal{P}''(q,f) = \begin{cases} 1 & \text{if } q = o \text{ and } (f = \texttt{x} \text{ or } f = \texttt{y}) \\ 0 & \text{otherwise} \end{cases}$$

Now, we define resources $\mathcal{R}, \mathcal{R}'$, and $\mathcal{R}''$ as follows:

$$\mathcal{R} \triangleq (h, \mathcal{P}) \qquad \mathcal{R}' \triangleq (h', \mathcal{P}') \qquad \mathcal{R}'' \triangleq (h'', \mathcal{P}'')$$

Intuitively, the three resources represent a state where there is only a `Point` at address o. Resource $\mathcal{R}$ represents a partial read-only point, resource $\mathcal{R}'$ represents a partial read-write point, and resource $\mathcal{R}''$ represents a complete read-write point. The following holds:

$$\mathcal{R} \leq \mathcal{R}' \leq \mathcal{R}''$$

3.2.4 Predicate Environments

The predicates that are declared in the class table define a predicate environment that maps predicate symbols to relations.

Predicate domains. What is the domain of these relations? Roughly, the domain consists of resources (including the heap) and tuples of specification values (representing class parameters and predicate parameters). To define this formally, first, let $\mathsf{SpecVals}$ be the set of all tuples of specification values:

$$\mathsf{SpecVals} \triangleq \bigcup_{n \geq 0} \mathsf{SpecVal}^n$$

Second, let $\mathsf{Pred}(ct)$ be the set of all qualified predicates $P\texttt{@}C$ that are defined in class table ct:

$$\mathsf{Pred}(ct) \triangleq \{\ P\texttt{@}C \mid C \in \mathsf{dom}(ct) \text{ and } P \text{ is defined in } C\ \}$$

For $P\texttt{@}C$ in $\mathsf{Pred}(ct)$, its domain $\mathsf{Dom}(P\texttt{@}C)$ is defined as the subset of $\mathsf{SpecVals} \times \mathsf{Resources} \times \mathsf{ObjId} \times \mathsf{SpecVals}$ that consists of all tuples $(\bar{\pi}, \mathcal{R}, r, \bar{\pi}')$ that satisfy the following conditions:

(a) $\mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash r : C\texttt{<}\bar{\pi}\texttt{>}$.

(b) $\mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>}) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}$ and $\mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash \bar{\pi}' : \bar{T}$ for some $\bar{T}, \bar{\alpha}$.

Intuitively, if $(\bar{\pi}, \mathcal{R}, r, \bar{\pi}') \in \mathsf{Dom}(P\texttt{@}C)$, then $\bar{\pi}$ represents the class parameters of r's dynamic class C, resource $\mathcal{R}$ represents the model used in the semantics relation (see Section 3.2.5), r represents the predicate receiver, and $\bar{\pi}'$ represents P's actual predicate parameters.

Predicate environments. We choose to represent relations as functions into the two-element set: Let 2 be the two-element set $\{0, 1\}$ equipped with the usual order (i.e., $0 \leq 1$). A *predicate environment* $\mathcal{E}$ is a function of type $\prod \kappa \in \mathsf{Pred}(ct).\mathsf{Dom}(\kappa) \to 2$ such that the following axiom holds:

(a) If $(\bar{\pi}, \mathcal{R}, r, \bar{\pi}'), (\bar{\pi}, \mathcal{R}', r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$ and $\mathcal{R} \leq \mathcal{R}'$,
then $\mathcal{E}(\kappa)(\bar{\pi}, \mathcal{R}, r, \bar{\pi}') \leq \mathcal{E}(\kappa)(\bar{\pi}, \mathcal{R}', r, \bar{\pi}')$.

Axiom (a) says that predicates are monotone in the resources: if a predicate is satisfied in resource $\mathcal{R}$, then it is also satisfied in all larger resources $\mathcal{R}'$. This axiom is natural for a language with garbage collection. As we extend the set of formulas in next chapters, we will extend the list of axioms that predicate environments must satisfy.

The class table's predicate environment. The class table ct defines a predicate environment that maps each predicate in ct to its definition. Technically, this predicate environment is defined as the least fixed point of the endofunction[2] $\mathcal{F}_{ct}$ on predicate environments. The definition of $\mathcal{F}_{ct}$ refers to the Kripke resource semantics $\models$, as defined in Section 3.2.5.

$$\frac{\begin{array}{l}\mathsf{pbody}(r.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}\texttt{>}) = F\ \texttt{ext}\ D\texttt{<}\bar{\pi}''\texttt{>} \\ C \neq \texttt{Object} \text{ and } \mathsf{arity}(P, D) = n \ \Rightarrow\ F' = r.P\texttt{@}D\texttt{<}\bar{\pi}'_{\mathsf{to}\,n}\texttt{>} \\ C = \texttt{Object} \text{ or } P \text{ is rooted in } C \ \Rightarrow\ F' = \texttt{true}\end{array}}{\mathcal{F}_{ct}(\mathcal{E})(P\texttt{@}C)(\bar{\pi}, \mathcal{R}, r, \bar{\pi}') = \begin{cases} 1 & \text{if } \mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash \mathcal{E}; \mathcal{R}; \emptyset \models F * F' \\ 0 & \text{otherwise}\end{cases}} \quad \text{(Sem Pred)}$$

In this definition, $\bar{\pi}'_{\mathsf{to}\,n}$ denotes the tuple consisting of the first n entries of $\bar{\pi}'$ (which may have more than n entries due to arity extension in subclasses).

Lemma 1 (Well-Typedness of $\mathcal{F}_{ct}$)**.** *If $\mathcal{E}$ is a predicate environment over $X \subseteq \mathsf{Pred}(ct)$, then so is $\mathcal{F}_{ct}(\mathcal{E})$.*

Proof. We need to show that $\mathcal{F}_{ct}(\mathcal{E})$ satisfies axiom (a) for predicate environments. But this is a consequence of Lemma 8 that we show in Section 6.1.1. □

Theorem 1 (Existence of Fixed Points)**.** *If $ct : \diamond$, then there exists a predicate environment $\mathcal{E}$ such that $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$.*

Proof. This is a consequence of Theorem 8 on fixed points in [54]. In the proof of this theorem, it is crucial to remark that, by syntactic restriction, cyclic predicate dependencies must be positive. □

[2] An endofunction is a function whose range is a subset of its domain.

3.2.5 Semantics

First, we define the semantics of expressions (recall that expressions of type `bool` are included in the domain of formulas). Because expressions contain specification values, we first give the semantics of specification values which consist of values and fractional permissions.

Because the semantics of values is the set of closed values ClVal, we use $\mathsf{SemVal} = \mathsf{ClVal} \cup (0, 1]$ to denote the semantics of specification values and we range over SemVal with meta-variable μ. Interval $(0, 1]$ represents the semantics of fractional permissions.

As specification values consist of values (whose semantics has been defined in Section 2.2, page 12) and fractional permissions, new definitions only cover the semantics of fractional permissions. We leave the semantics of logical variables α undefined, because we deal with these variables by substitution:

Semantics of Specification Values, $[\![.]\!] : \mathsf{SpecVal} \to \mathsf{SemVal}$:

$$\ldots \quad [\![1]\!] \triangleq 1 \quad [\![\mathtt{split}(\pi)]\!] \triangleq \tfrac{1}{2} \cdot [\![\pi]\!] \quad \ldots$$

We define the semantics of expressions:

Semantics of Expressions, $[\![e]\!] : \mathsf{Heap} \to \mathsf{Stack} \rightharpoonup \mathsf{SemVal}$:

$$\text{(Sem SpecVal)}\ \frac{[\![\pi]\!] = \mu}{[\![\pi]\!]^h_s = \mu} \qquad \text{(Sem Var)}\ \frac{s(\ell) = v}{[\![\ell]\!]^h_s = v} \qquad \text{(Sem Op)}\ \frac{[\![w_1]\!]^h_s = v_1 \ \cdots\ [\![w_n]\!]^h_s = v_n \quad [\![op]\!]^h(v_1, \ldots, v_n) = v}{[\![op(w_1, \ldots, w_n)]\!]^h_s = v}$$

Now, we define the semantics of formulas. Let $(\Gamma \vdash \mathcal{R} : \diamond)$ whenever $\Gamma \vdash \mathcal{R}_{\mathsf{hp}} : \diamond$ and $\mathcal{P}(o, f) > 0$ implies $o \in \mathsf{dom}(\Gamma)$. Furthermore, let $(\Gamma \vdash \mathcal{E}, \mathcal{R}, s, F : \diamond)$ whenever $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$, $\Gamma \vdash \mathcal{R} : \diamond$, $\Gamma \vdash s : \diamond$, and $\Gamma \vdash F : \diamond$. The relation $(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F)$ is the unique subset of $(\Gamma \vdash \mathcal{E}, \mathcal{R}, s, F : \diamond)$ that satisfies the following clauses:

$$
\begin{array}{rcll}
\Gamma \vdash (h,\mathcal{P}); s & \models & e & \text{iff } [\![e]\!]^h_s = \texttt{true} \\
\Gamma \vdash \mathcal{E}; (h,\mathcal{P}); s & \models & \texttt{PointsTo}(e.f,\pi,e') & \text{iff } \begin{cases} [\![e]\!]^h_s = o, h(o)_2(f) = [\![e']\!]^h_s \text{ and} \\ [\![\pi]\!] \leq \mathcal{P}(o,f) \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>} & \text{iff true} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} & \text{iff } \begin{cases} \mathcal{R}_{\mathsf{hp}}(o)_1 <: C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathcal{E}(P\texttt{@}C)(\bar{\pi}', \mathcal{R}, o, \bar{\pi}) = 1 \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & o.P\texttt{<}\bar{\pi}\texttt{>} & \text{iff } \begin{cases} (\exists \bar{\pi}'')(\mathcal{R}_{\mathsf{hp}}(o)_1 = C\texttt{<}\bar{\pi}'\texttt{>} \text{ and} \\ \mathcal{E}(P\texttt{@}C)(\bar{\pi}', \mathcal{R}, o, (\bar{\pi}, \bar{\pi}'')) = 1) \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F * G & \text{iff } \begin{cases} (\exists \mathcal{R}_1, \mathcal{R}_2)(\mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2, \\ \Gamma \vdash \mathcal{E}; \mathcal{R}_1; s \models F \text{ and } \Gamma \vdash \mathcal{E}; \mathcal{R}_2; s \models G) \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \texttt{-*} G & \text{iff } \begin{cases} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathcal{R}')(\\ \mathcal{R} \# \mathcal{R}' \text{ and } \Gamma' \vdash \mathcal{E}; \mathcal{R}'; s \models F \\ \Rightarrow\ \Gamma' \vdash \mathcal{E}; \mathcal{R} * \mathcal{R}'; s \models G\ ) \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \texttt{\&} G & \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F \text{ and } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models G \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & F \texttt{|} G & \text{iff } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F \text{ or } \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models G \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & \texttt{(ex } T\ \alpha\texttt{)(}F\texttt{)} & \text{iff } \begin{cases} (\exists \pi)(\ \Gamma_{\mathsf{hp}} \vdash \pi : T \text{ and} \\ \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F[\pi/\alpha]\ ) \end{cases} \\
\Gamma \vdash \mathcal{E}; \mathcal{R}; s & \models & \texttt{(fa } T\ \alpha\texttt{)(}F\texttt{)} & \text{iff } \begin{cases} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathcal{R}' \geq \mathcal{R}, \pi)(\\ \Gamma'_{\mathsf{hp}} \vdash \mathcal{R}'_{\mathsf{hp}} : \diamond \text{ and } \Gamma'_{\mathsf{hp}} \vdash \pi : T \\ \Rightarrow\ \Gamma' \vdash \mathcal{E}; \mathcal{R}'; s \models F[\pi/\alpha]\ ) \end{cases}
\end{array}
$$

Example. We exemplify relation $\models$ by reusing Figure 2.1's class `2DPoint`. We add predicate `state` to class `2DPoint`:

```
class 2DPoint{
  int x;
  int y;
  pred state<perm p> = (ex int i,j)(PointsTo(x,p,i) * PointsTo(y,p,j));
}
```

Let ct be the class table that only contains class `2DPoint` and $\mathcal{E}$ be some predicate environment such that $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$ (which exists by Theorem 1). We define Γ, h, and $\mathcal{P}$ as follows:

$$
\Gamma \triangleq o \mapsto \texttt{2DPoint} \qquad h \triangleq o \mapsto (\texttt{2DPoint}, \texttt{x} \mapsto 0, \texttt{y} \mapsto 0)
$$

$$
\mathcal{P}(q)(f) = \begin{cases} 1 & \text{if } q = o \text{ and } (f = \texttt{x} \text{ or } f = \texttt{y}) \\ 0 & \text{otherwise} \end{cases}
$$

Now, we show that $\Gamma \vdash \mathcal{E}; (h,\mathcal{P}); \emptyset \models o.\texttt{state<}\frac{1}{2}\texttt{>} * \texttt{PointsTo}(o.\texttt{x}, \frac{1}{2}, 1)$ holds:

Proof. By the case $*$ of the semantics, we have to find $h', \mathcal{P}', h''$ and $\mathcal{P}''$ such that the following statements hold:

(a) $\Gamma \vdash \mathcal{E}; (h', \mathcal{P}'); \emptyset \models o.\texttt{state<}\frac{1}{2}\texttt{>}$ (goal)
(b) $\Gamma \vdash \mathcal{E}; (h'', \mathcal{P}''); \emptyset \models \texttt{PointsTo}(o.\texttt{x}, \frac{1}{2}, 1)$ (goal)

(c) $(h', \mathcal{P}') * (h'', \mathcal{P}'') = (h, \mathcal{P})$ (goal)

We define:

$$h' = h'' = h \quad \mathcal{P}'(q)(f) = \mathcal{P}''(q, f) = \begin{cases} 1/2 & \text{if } q = o \text{ and } (f = \mathtt{x} \text{ or } f = \mathtt{y}) \\ 0 & \text{otherwise} \end{cases} \qquad \mathcal{R} \triangleq (h, \mathcal{P})$$

Because $h' = h'' = h$ and $(\forall o, f)(\mathcal{P}'(o)(f) + \mathcal{P}''(o, f) = \mathcal{P}(o, f))$, goal (c) is closed. By the case for unqualified predicates of our semantics, to show goal (a), we need to show: $\mathcal{E}(\mathtt{state@2DPoint})(\emptyset, (h', \mathcal{P}'), o, \frac{1}{2})$. By applying rule (Sem Pred), it suffices to show: $\Gamma \vdash \mathcal{E}; (h', \mathcal{P}'); \emptyset \models \mathtt{PointsTo}(o.\mathtt{x}, \frac{1}{2}, \mathtt{1}) * \mathtt{PointsTo}(o.\mathtt{y}, \frac{1}{2}, \mathtt{1})$. By the case $*$ of the semantics, we have to find $h'_1, \mathcal{P}'_1, h'_2$ and $\mathcal{P}'_2$ such that the following statements hold:

(d) $\Gamma \vdash \mathcal{E}; (h'_1, \mathcal{P}'_1); \emptyset \models \mathtt{PointsTo}(o.\mathtt{x}, \frac{1}{2}, \mathtt{1})$
(e) $\Gamma \vdash \mathcal{E}; (h'_2, \mathcal{P}'_2); \emptyset \models \mathtt{PointsTo}(o.\mathtt{y}, \frac{1}{2}, \mathtt{1})$
(f) $(h'_1, \mathcal{P}'_1) * (h'_2, \mathcal{P}'_2) = (h', \mathcal{P}')$

We define:

$$h'_1 = o \mapsto (\mathtt{2DPoint}, \mathtt{x} \mapsto \mathtt{0}) \quad \mathcal{P}'_1(q)(f) = \begin{cases} 1/2 & \text{if } q = o \text{ and } f = \mathtt{x} \\ 0 & \text{otherwise} \end{cases}$$
$$h'_2 = o \mapsto (\mathtt{2DPoint}, \mathtt{y} \mapsto \mathtt{0}) \quad \mathcal{P}'_2(q)(f) = \begin{cases} 1/2 & \text{if } q = o \text{ and } f = \mathtt{y} \\ 0 & \text{otherwise} \end{cases}$$

Note that h'_1 and $\mathcal{P}'_1$ are such that $(h'_1, \mathcal{P}'_1)$ satisfies Section 3.2.3's resource axioms (a) and (b) (a similar remark applies for $h'_2, \mathcal{P}'_2$ and $(h'_2, \mathcal{P}'_2)$). Because $h'_1 * h'_2 = h'$ and $\mathcal{P}'_1 * \mathcal{P}'_2 = \mathcal{P}'$, goal (f) is closed. Now, by the case $\mathtt{PointsTo}$ of our semantics, $\mathcal{P}'_1(o, \mathtt{x}) = 1/2$, and $\mathcal{P}'_2(o, \mathtt{y}) = 1/2$, goals (d) and (e) are closed. This closes goal (a).

Goal (b) is routine. □

Difference between Intuitionistic and Classical Separation Logic. Intuitionistic and classical separation logic differ on the semantics of the $\mathtt{PointsTo}$ predicate. In intuitionistic separation logic, the formula $\mathtt{PointsTo}(e.f, \pi, e')$ asserts there is *at least* one allocated cell at the address pointed to by $e.f$; while in classical separation logic, the formula $\mathtt{PointsTo}(e.f, \pi, e')$ asserts there is *exactly* one allocated cell at the address pointed to by $e.f$. Consequently, the intuitionistic semantics admits weakening. This is appropriate to reason about garbage collected languages such as Java. Intuitively, wherever some object can be garbaged, it is sound to forget this object's state by using weakening. Dually, in languages where deallocation is done by programmers (such as C and C++), the state of deallocated objects is consumed by the primitive for deallocation.

3.2.6 Proof Theory

As usual, Hoare triples will be based on a logical consequence judgment. We define logical consequence proof-theoretically. The proof theory has two judgments:

$\Gamma; v; \bar{F} \vdash G$ G is a logical consequence of the $*$-conjunction of $\bar{F}$
$\Gamma; v \vdash F$ F is an axiom

In the former judgment, $\bar{F}$ is a *multiset* of formulas. The parameter v represents the *current receiver*. The receiver parameter is needed to determine the scope of predicate definitions: a receiver v knows the definitions of predicates of the form $v.P$, but not

the definitions of other predicates. In source code verification, the receiver parameter is always `this` and can thus be omitted. We explicitly include the receiver parameter in the general judgment, because we want the proof theory to be closed under value substitutions.

Semantic Validity of Boolean Expressions. The proof theory depends on the relation $\Gamma \models e$ ("e is valid in all well-typed heaps"), which we do not axiomatize (in an implementation, we would use external and dedicated theorem provers to decide this relation). To define this relation, let σ range over *closing substitutions*, i.e, elements of $\mathsf{Var} \rightharpoonup \mathsf{ClVal}$.

$$\frac{\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma) \cap \mathsf{Var} \quad (\forall x \in \mathsf{dom}(\sigma))(\Gamma_{\mathsf{hp}} \vdash \sigma(x) : \Gamma(x)[\sigma])}{\Gamma \vdash \sigma : \diamond}$$

$$\mathsf{ClosingSubst}(\Gamma) \triangleq \{\ \sigma \mid \Gamma \vdash \sigma : \diamond\ \}$$

We say that a heap h is *total* iff for all o in $\mathsf{dom}(h)$ and all $f \in \mathsf{dom}(\mathsf{fld}(h(o)_1))$ it is the case that $f \in \mathsf{dom}(h(o)_2)$ (remember that heaps were defined on page 12).

$$\mathsf{Heap}(\Gamma) \triangleq \{\ h \mid \Gamma_{\mathsf{hp}} \vdash h : \diamond \text{ and } h \text{ is total}\ \}$$

Now, we define $\Gamma \models e$ as follows:

$$\Gamma \models e \quad \text{iff} \quad \begin{cases} \Gamma \vdash e : \texttt{bool} \text{ and} \\ (\ \forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, h \in \mathsf{Heap}(\Gamma'), \sigma \in \mathsf{ClosingSubst}(\Gamma')\ )(\ [\![e[\sigma]]\!]^h_\emptyset = \texttt{true}\ ) \end{cases}$$

Natural Deduction Rules. The logical consequence judgment of our Hoare logic is based on the natural deduction calculus of *(affine) linear logic* [99], which coincides with BI's natural deduction calculus [85] on our restricted set of logical operators. To avoid a proof theory with bunched contexts, we omit the $\Rightarrow$-implication between heap formulas (and did not need it in later examples). However, this design decision is not essential.

Logical Consequence, $\Gamma; v; \bar{F} \vdash G$:

(Id) $$\frac{\Gamma \vdash v, \bar{F}, G : \texttt{Object}, \diamond}{\Gamma; v; \bar{F}, G \vdash G}$$

(Ax) $$\frac{\Gamma; v \vdash G \quad \Gamma \vdash v, \bar{F}, G : \texttt{Object}, \diamond}{\Gamma; v; \bar{F} \vdash G}$$

($*$ Intro) $$\frac{\Gamma; v; \bar{F} \vdash H_1 \quad \Gamma; v; \bar{G} \vdash H_2}{\Gamma; v; \bar{F}, \bar{G} \vdash H_1 * H_2}$$

($*$ Elim) $$\frac{\Gamma; v; \bar{F} \vdash G_1 * G_2 \quad \Gamma; v; \bar{E}, G_1, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$$

(-$*$ Intro) $$\frac{\Gamma; v; \bar{F}, G_1 \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mathrel{-\!\!*} G_2}$$

(-$*$ Elim) $$\frac{\Gamma; v; \bar{F} \vdash H_1 \mathrel{-\!\!*} H_2 \quad \Gamma; v; \bar{G} \vdash H_1}{\Gamma; v; \bar{F}, \bar{G} \vdash H_2}$$

(& Intro) $$\frac{\Gamma; v; \bar{F} \vdash G_1 \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}$$

(& Elim 1) $$\frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_1}$$

(& Elim 2) $$\frac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_2}$$

(| Intro 1) $$\frac{\Gamma; v; \bar{F} \vdash G_1}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$$

(| Intro 2) $$\frac{\Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$$

(| Elim) $$\frac{\Gamma; v; \bar{F} \vdash G_1 \mid G_2 \quad \Gamma; v; \bar{E}, G_1 \vdash H \quad \Gamma; v; \bar{E}, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$$

$$\text{(Ex Intro)}\quad \frac{\Gamma, \alpha : T \vdash G : \diamond \quad \Gamma \vdash \pi : T \quad \Gamma; v; \bar{F} \vdash G[\pi/\alpha]}{\Gamma; v; \bar{F} \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)}} \qquad \text{(Ex Elim)}\quad \frac{\alpha \notin \bar{F}, H \quad \Gamma; v; \bar{E} \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)} \quad \Gamma, \alpha : T; v; \bar{F}, G \vdash H}{\Gamma; v; \bar{E}, \bar{F} \vdash H}$$

$$\text{(Fa Intro)}\quad \frac{\alpha \notin \bar{F} \quad \Gamma, \alpha : T; v; \bar{F} \vdash G}{\Gamma; v; \bar{F} \vdash \texttt{(fa}\ T\ \alpha\texttt{)(}G\texttt{)}} \qquad \text{(Fa Elim)}\quad \frac{\Gamma; v; \bar{F} \vdash \texttt{(fa}\ T\ \alpha\texttt{)(}G\texttt{)} \quad \Gamma \vdash \pi : T}{\Gamma; v; \bar{F} \vdash G[\pi/\alpha]}$$

The proof system above has been implemented in a prototype tool called *ratp* [62]. The implementation was used to verify entailment between formulas when verifying example programs shown later.

Axioms. In addition to the logical consequence defined above, sound *axioms* capture additional properties of our model. By axioms, we mean that they can be added to our logical consequence judgment without harming soundness. This is shown by Theorem 2 below. We now present these axioms.

The first axiom regulates permission accounting (where v denotes the current receiver and $\frac{\pi}{2}$ abbreviates $\texttt{split}(\pi)$):

(Split/Merge)
$$\Gamma; v \vdash \texttt{PointsTo}(e.f, \pi, e') \texttt{ *-* (}\texttt{PointsTo}(e.f, \tfrac{\pi}{2}, e') \texttt{ * }\texttt{PointsTo}(e.f, \tfrac{\pi}{2}, e')\texttt{)}$$

The next axiom allows predicate receivers to toggle between predicate names and predicate definitions (where – as defined in Appendix A – $\texttt{pbody}(o.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}\texttt{>})$ looks up $o.P\texttt{<}\bar{\pi}'\texttt{>}$'s definition in the type $C\texttt{<}\bar{\pi}\texttt{>}$ and returns its body F together with $C\texttt{<}\bar{\pi}\texttt{>}$'s direct superclass $D\texttt{<}\bar{\pi}''\texttt{>}$):

$$(\Gamma \vdash v : C\texttt{<}\bar{\pi}''\texttt{>} \ \wedge\ \texttt{pbody}(v.P\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F\ \texttt{ext}\ D\texttt{<}\bar{\pi}'''\texttt{>}) \Rightarrow\ \Gamma; v \vdash v.P\texttt{@}C\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>}\ \texttt{*-*}\ \texttt{(}F\ \texttt{*}\ v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>)} \qquad \text{(Open/Close)}$$

Note that the current receiver, as represented on the left of the $\vdash$, has to match the predicate receiver on the right. This rule is the only reason why our logical consequence judgment tracks the current receiver. Note also that $P\texttt{@}C$ may have more parameters than $P\texttt{@}D$: following Parkinson [88] we allow subclasses to extend predicate arities. Missing predicate parameters are existentially quantified, as expressed by the following axiom:

$$\Gamma; v \vdash \pi.P\texttt{<}\bar{\pi}\texttt{>}\ \texttt{*-*}\ \texttt{(ex}\ \bar{T}\ \bar{\alpha}\texttt{)(}\pi.P\texttt{<}\bar{\pi}, \bar{\alpha}\texttt{>)} \qquad \text{(Missing Parameters)}$$

The following axiom says that a predicate at a receiver's dynamic type (i.e., without @-selector) is stronger than the predicate at its static type. In combination with (Open/Close), this allows to open and close predicates at the receiver's static type:

$$\Gamma; v \vdash \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Dynamic Type)}$$

There is another similar axiom:

$$C \preceq D \ \Rightarrow\ \Gamma; v \vdash \pi.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{@}C\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>} \qquad (\texttt{ispartof}\ \text{Monotonic})$$

The following axiom allows to drop the class modifier C from $\pi.P\texttt{@}C$ if we know that C is π's dynamic class:

$$\Gamma; v \vdash \texttt{(}\ \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{*}\ C\ \texttt{classof}\ \pi\ \texttt{)}\ \texttt{-*}\ \pi.P\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Known Type)}$$

Our semantics of predicates defines predicates with `null`-receiver to hold:

$$\Gamma; v \vdash \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>} \qquad \text{(Null Receiver)}$$

We axiomatize `true` and `false`:

$$\Gamma; v \vdash \texttt{true} \qquad \text{(True)} \qquad\qquad \Gamma; v \vdash \texttt{false}\ \texttt{-*}\ F \qquad \text{(False)}$$

The substitutivity axiom allows to replace expressions by equal expressions:

$$(\Gamma \vdash e, e' : T\ \wedge\ \Gamma, x : T \vdash F : \diamond)\ \Rightarrow\ \Gamma; v \vdash \texttt{(}F[e/x]\ \texttt{*}\ e\ \texttt{==}\ e'\texttt{)}\ \texttt{-*}\ F[e'/x]$$

The next axiom lifts semantic validity of boolean expressions to the proof theory:

$$(\Gamma \models \texttt{!}e_1\ \texttt{|}\ \texttt{!}e_2\ \texttt{|}\ e')\ \Rightarrow\ \Gamma; v \vdash \texttt{(}e_1\ \texttt{*}\ e_2\texttt{)}\ \texttt{-*}\ e'$$

The following axiom captures that fields point to a unique value. Recall that we write "F `assures` G" to abbreviate "F `-*` `(`F `*` G`)`" (see Section 3.2.1):

$$\Gamma; v \vdash \texttt{(PointsTo(}e.f, \pi, e'\texttt{)}\ \texttt{\&}\ \texttt{PointsTo(}e.f, \pi', e''\texttt{))}\ \texttt{assures}\ e'\ \texttt{==}\ e''$$

Then, there is an axiom that captures that all well-typed closed expressions represent a value (because built-in operations are total):

$$(\Gamma \vdash e : T)\ \Rightarrow\ \Gamma; v \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}e\ \texttt{==}\ \alpha\texttt{)}$$

Finally, there is a copyability axiom for boolean expressions:

$$\Gamma; v \vdash \texttt{(}F\ \texttt{\&}\ e\texttt{)}\ \texttt{-*}\ \texttt{(}F\ \texttt{*}\ e\texttt{)} \qquad \text{(Copyable)}$$

Soundness of the proof theory. We define semantic entailment $\Gamma \vdash \mathcal{E}; \bar{F} \models G$:

$$\begin{array}{rlll} \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F_1, \ldots, F_n & \text{iff} & \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F_1 \texttt{*} \cdots \texttt{*} F_n \\ \Gamma \vdash \mathcal{E}; \bar{F} \models G & \text{iff} & (\forall \Gamma, \mathcal{R}, s)(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models \bar{F}\ \Rightarrow\ \Gamma \vdash \mathcal{E}; \mathcal{R}; s \models G) \end{array}$$

Now, we can express the proof theory's soundness:

Theorem 2 (Soundness of Logical Consequence)**.** *If* $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$ *and* $(\Gamma; o; \bar{F} \vdash G)$*, then* $(\Gamma \vdash \mathcal{E}; \bar{F} \models G)$*.*

Proof. The proof is by induction on $(\Gamma; o; \bar{F} \vdash G)$'s proof tree. The pen and paper proof can be found in [54, §R]. The proof has been mechanically checked for a smaller specification language (without abstract predicates and quantifiers) [62]. □

3.3 Hoare Triples

In this section, we present Hoare rules to verify programs written in Section 2.1's language.

Hoare triples for *head commands* have the following form:

$$\Gamma; v \vdash \{F\} hc \{G\}$$

First, we present the rule for field writing . The rule's precondition[3] requires that the heap contains at least the object dereferenced and the field mentioned. In addition, it requires permission 1 to this object's field, i.e., write-permission. The rule's postcondition simply ensures that the heap has been updated with the value assigned. It should be noted that this rule is *small* [86]: it does not require anything more than a single `PointsTo` predicate. The (Frame) rule (discussed below) is used to build proofs in bigger contexts.

$$\frac{\Gamma \vdash u, w : U, W \quad W\ f \in \mathsf{fld}(U)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f, 1, W)\} u.f = w \{\texttt{PointsTo}(u.f, 1, w)\}} \text{ (Fld Set)}$$

The rule for field reading requires a `PointsTo` predicate with *any* permission π:

$$\frac{\Gamma \vdash u, \pi, w : U, \texttt{perm}, W \quad W\ f \in \mathsf{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f, \pi, w)\} \ell = u.f \{\texttt{PointsTo}(u.f, \pi, w) * \ell == w\}} \text{ (Get)}$$

The rule for creating new objects has `true` as a precondition. That is because we do not check for out of memory errors. After creating an object, all its fields are writable: the ℓ.`init` predicate (formally defined in Appendix A) `*`-conjoins the predicates $\texttt{PointsTo}(\ell.f, 1, \mathsf{df}(T))$ for all fields $T\ f$ in ℓ's class:

$$\frac{C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{true}\} \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>} \{\ell.\texttt{init} * C\ \texttt{classof}\ \ell\}} \text{ (New)}$$

The rule for method calls is verbose, but standard:

$$\frac{\begin{array}{c}\mathsf{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ G; \texttt{ensures}\ (\texttt{ex}\ U\ \alpha')(G');\ U\ m(t\texttt{<}\bar{\pi}\texttt{>}\ \iota_0; \bar{W}\ \bar{\iota}) \\ \sigma = (u/\iota_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\iota}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t\texttt{<}\bar{\pi}\texttt{>}, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma; v \vdash \{u\ \texttt{!=}\ \texttt{null} * G[\sigma]\} \ell = u.m(\bar{w}) \{(\texttt{ex}\ U[\sigma]\ \alpha')(\alpha' == \ell * G'[\sigma])\}} \text{ (Call)}$$

Figure 3.1 lists the remaining standard rules, including rules for commands. Our judgment for *commands* combines typing and Hoare triples:

$$\Gamma; v \vdash \{F\} c : T \{G\}$$

T is a type of the return value (possibly a supertype of the return value's dynamic type). G is the postcondition and is always of the form $G = (\texttt{ex}\ U\ \alpha)(G')$ with $U <: T$. The existentially quantified α represents the return value. Figure 3.1 shows a rule for `assert` statements which are defined later (on page 41). Even though `return` statements are not available to programmers, Figure 3.1 shows a rule for `return` statements because we added this auxiliary syntax in Section 2.2.

Importantly, our system includes the (Frame) rule. To understand this rule, note that $\mathsf{fv}(F)$ is the set of free variables of F and that we write $x \notin F$ to abbreviate $x \notin \mathsf{fv}(F)$. Furthermore, we write $\mathsf{writes}(hc)$ for the set of read-write variables ℓ that occur freely on the left-hand-side of an assignment in hc. (Frame)'s side condition on variables is standard [84, 88]. Bornat showed how to get rid of this side condition [20] by treating variable as resources.

[3] Where, as defined in Section 3.2.1, $\texttt{PointsTo}(u.f, 1, W)$ stands for $(\texttt{ex}\ W\ w)(\texttt{PointsTo}(u.f, 1, w))$.

$$\frac{\Gamma; v; F \vdash G[w/\alpha] \quad \Gamma \vdash w : U <: T \quad \Gamma, \alpha : U \vdash G : \diamond}{\Gamma; v \vdash \{F\} w : T\{(\mathtt{ex}\ U\ \alpha)(G)\}} \text{ (Val)}$$

$$\frac{\ell \notin F, G \quad \Gamma, \ell : T; v \vdash \{F * \ell == \mathsf{df}(T)\} c : U\{G\}}{\Gamma; v \vdash \{F\} T\ \ell;\ c : U\{G\}} \text{ (Dcl)}$$

$$\frac{\begin{array}{l}\imath \notin F, G, v \\ \Gamma \vdash \ell : T \quad \Gamma, \imath : T; v \vdash \{F * \imath == \ell\} c : U\{G\}\end{array}}{\Gamma; v \vdash \{F\} T\ \imath = \ell;\ c : U\{G\}} \text{ (Fin Dcl)}$$

$$\frac{\Gamma; v \vdash \{F\} hc \{F'\} \quad \Gamma; v \vdash \{F'\} c : T\{G\}}{\Gamma; v \vdash \{F\} hc; c : T\{G\}} \text{ (Seq)}$$

$$\frac{\Gamma; v \vdash \{F\} hc\{G\} \quad \Gamma \vdash H : \diamond \quad \mathsf{fv}(H) \cap \mathsf{writes}(hc) = \emptyset}{\Gamma; v \vdash \{F * H\} hc \{G * H\}} \text{ (Frame)}$$

$$\frac{\begin{array}{l}\Gamma; v \vdash \{F'\} hc \{G'\} \\ \Gamma; v; F \vdash F' \quad \Gamma; v; G' \vdash G\end{array}}{\Gamma; v \vdash \{F\} hc \{G\}} \text{ (Consequence)} \qquad \frac{\Gamma, \alpha : T; v \vdash \{F\} hc \{G\}}{\Gamma; v \vdash \{(\mathtt{ex}\ T\ \alpha)(F)\} hc \{(\mathtt{ex}\ T\ \alpha)(G)\}} \text{ (Exists)}$$

$$\frac{\Gamma \vdash w : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = w \{\ell == w\}} \text{ (Var Set)} \qquad \frac{\Gamma \vdash op(\bar{w}) : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = op(\bar{w}) \{\ell == op(\bar{w})\}} \text{ (Op)}$$

$$\frac{\begin{array}{l}\Gamma \vdash w : \mathtt{bool} \\ \Gamma; v \vdash \{F * w\} c : \mathtt{void}\{G\} \quad \Gamma; v \vdash \{F * !w\} c' : \mathtt{void}\{G\}\end{array}}{\Gamma; v \vdash \{F\} \mathtt{if}\ (w) \{c\} \mathtt{else} \{c'\} \{G\}} \text{ (If)}$$

$$\frac{\Gamma; v; F \vdash G}{\Gamma; v \vdash \{F\} \mathtt{assert}(G) \{F\}} \text{ (Assert)}$$

$$\frac{\Gamma \vdash v : T \quad \Gamma; o; F \vdash G[v/\alpha] \quad T <: U \quad \Gamma, \ell : U; p \vdash \{(\mathtt{ex}\ T\ \alpha)(\alpha == \ell\ *\ G)\} c : V\{H\}}{\Gamma, \ell : U; o \vdash \{F\} \ell = \mathtt{return}(v);\ c : V\{H\}} \text{ (Return)}$$

Figure 3.1: Hoare triples

3.4 Verified Interfaces and Classes

Before defining a judgment for verified programs, we need to define judgments for verified interfaces and classes. To do this, we first define method subtyping and predicate subtyping.

Method Subtyping. First, recall that *method types* are of the following form:

$$\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ F; \texttt{ensures}\ G; U\ m\texttt{(}V_0\ \imath_0; \bar{V}\ \bar{\imath}\texttt{)}$$

The self-parameter ($\imath_0$) is explicit, separated from the other formal parameters by a semicolon. Before presenting the method subtyping rule in full generality, we present its instance for method types without logical parameters:

$$\frac{U, V_0, \bar{V}' <: U', V_0', \bar{V} \quad \Gamma, \imath_0 : V_0, \bar{\imath} : \bar{V}';\ \imath_0;\ \texttt{true} \vdash F' \texttt{-*}\ \texttt{(}F \texttt{*}\ \texttt{(fa}\ U\ \texttt{result)(}G \texttt{-*}\ G'\texttt{))}}{\Gamma \vdash \texttt{requires}\ F; \texttt{ensures}\ G; U\ m\texttt{(}V_0\ \imath_0; \bar{V}\ \bar{\imath}\texttt{)} <: \texttt{requires}\ F'; \texttt{ensures}\ G'; U'\ m\texttt{(}V_0'\ \imath_0; \bar{V}'\ \bar{\imath}\texttt{)}}$$

To understand this rule, we invite the reader to consider the following two derived rules (where types are elided):

$$\frac{\vdash F' \texttt{-*}\ F \quad \vdash G \texttt{-*}\ G'}{\vdash \texttt{requires}\ F; \texttt{ensures}\ G <: \texttt{requires}\ F'; \texttt{ensures}\ G'}$$

$$\frac{}{\vdash \texttt{requires}\ F; \texttt{ensures}\ G <: \texttt{requires}\ F \texttt{*}\ H; \texttt{ensures}\ G \texttt{*}\ H}$$

The first of these derived rules is standard behavioral subtyping, the second one abstracts separation logic's frame rule. In order to see that these two rules follow from the above rule, note that the following two formulas are tautologies (as can be easily proven by natural deduction):

$$\texttt{(}F' \texttt{-*}\ F\texttt{)} \texttt{*}\ H \texttt{-*}\ F' \texttt{-*}\ F \texttt{*}\ H \qquad F \texttt{*}\ H \texttt{-*}\ F \texttt{*}\ \texttt{(fa}\ U\ x\texttt{)(}G \texttt{-*}\ G \texttt{*}\ H\texttt{)}$$

The general method subtyping rule also accounts for logical parameters (where we abbreviate `requires` and `ensures`):

$$\frac{\Gamma, \imath_0 : V_0; \imath_0; \texttt{true} \vdash \texttt{(fa}\ \bar{T}'\ \bar{\alpha}\texttt{)(fa}\ \bar{V}'\ \bar{\imath}\texttt{)(}F' \texttt{-*}\ \texttt{(ex}\ \bar{W}\ \bar{\alpha}'\texttt{)(}F \texttt{*}\ \texttt{(fa}\ U\ \texttt{result)(}G \texttt{-*}\ G'\texttt{)))}}{\Gamma \vdash \texttt{<}\bar{T}\ \bar{\alpha}, \bar{W}\ \bar{\alpha}'\texttt{>req}\ F; \texttt{ens}\ G; U\ m\texttt{(}V_0\ \imath_0; \bar{V}\ \bar{\imath}\texttt{)} <: \texttt{<}\bar{T}'\ \bar{\alpha}\texttt{>req}\ F'; \texttt{ens}\ G'; U'\ m\texttt{(}V_0'\ \imath_0; \bar{V}'\ \bar{\imath}\texttt{)}}$$

Note that the subtype may have more logical parameters than the supertype. For instance, we obtain the following derived rule:

$$\frac{}{\vdash \texttt{<}T\ \alpha\texttt{>requires}\ F; \texttt{ensures}\ G <: \texttt{requires (ex}\ T\ \alpha\texttt{)(}F\texttt{)}; \texttt{ensures (ex}\ T\ \alpha\texttt{)(}G\texttt{)}}$$

This derived rule is an abstraction of separation logic's auxiliary variable rule (Exists) (see page 36). It follows from the method subtyping rule by the following tautology:

$$\texttt{(ex}\ T\ \alpha\texttt{)(}F\texttt{)} \texttt{-*}\ \texttt{(ex}\ T\ \alpha\texttt{)(}F \texttt{*}\ \texttt{(fa}\ U\ x\texttt{)(}G \texttt{-*}\ \texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)))}$$

Predicate Subtyping. Predicate type pt is a subtype of pt', if pt and pt' have the same name and pt's parameter signature "extends" pt''s parameter signature:

$$\frac{}{\texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}, \bar{T}'\ \bar{\alpha}'\texttt{>} <: \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}}$$

Class Axioms. Recall that we use `axioms` to export useful relations between predicates to clients. For this to be sound, programmers have to prove axioms sound. We require that class axioms are proven sound with a restricted logical consequence judgment:

$$\vdash' \quad \triangleq \quad \vdash \text{ without class axioms}$$

We disallow the application of class axioms for proving class axioms in order to avoid circularities. A class is sound if all its axioms are sound (the lookup function for axioms (axiom) is defined in Appendix A):

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{sound}$$
$$\text{iff}$$
$$\mathsf{axiom}(C\texttt{<}\bar{\alpha}\texttt{>}) = F \ \Rightarrow\ \bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>};\ \texttt{this};\ C\ \texttt{classof this} \vdash' F$$

Class Extensions and Interface Implementations. To define sanity conditions on classes and interfaces, we define some lookup functions.

$$\begin{array}{rcl}
\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}} & \rightarrow & \mathsf{methods}(C) \triangleq \mathsf{dom}(md\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}} & \rightarrow & \mathsf{methods}(I) \triangleq \mathsf{dom}(mt\texttt{*}) \\
\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}} & \rightarrow & \mathsf{preds}(C) \triangleq \mathsf{dom}(pd\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}} & \rightarrow & \mathsf{preds}(I) \triangleq \mathsf{dom}(pt\texttt{*}) \\
\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}} & \rightarrow & \mathsf{declared}(C) \triangleq \mathsf{dom}(fd\texttt{*})
\end{array}$$

Now, we define sanity conditions on classes and interfaces. These conditions are later used to ensure that we only verify sane programs.

In the definitions below, we conceive the partial functions mtype and ptype (which are formally defined in Appendix A) as total functions that map elements outside their domains to the special element undef. Furthermore, we extend the subtyping relation: $<: = \{(T, U) \mid T <: U\} \cup \{(\mathsf{undef}, \mathsf{undef})\}$.

Judgment $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U$ expresses that: (1) class C extends another class U, (2) class C does not redeclare inherited fields, and (3) methods and predicates overridden in class C are subtypes of the corresponding methods and predicates implemented in class U:

$$C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U \triangleq \begin{cases}
U \text{ is a parameterized class} \\
f \in \mathsf{dom}(\mathsf{fld}(U)) \ \Rightarrow\ f \notin \mathsf{declared}(C) \\
(\forall m, mt)(\mathsf{mtype}(m, U) = MT \ \Rightarrow\ \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, C\texttt{<}\bar{\alpha}\texttt{>}) <: MT) \\
(\forall P, pt)(\mathsf{ptype}(P, U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\bar{\alpha}\texttt{>}) <: pt)
\end{cases}$$

Judgment $I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ U$ expresses that: (1) interface I extends another interface U and (2) methods and predicates overridden in interface I are subtypes of the corresponding methods and predicates declared in U.

$$I\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{type\text{-}extends}\; U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \;\Rightarrow \\ \qquad\qquad \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, I\langle\bar{\alpha}\rangle) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \;\Rightarrow\; \mathsf{ptype}(P, I\langle\bar{\alpha}\rangle) <: pt) \end{cases}$$

$$I\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{type\text{-}extends}\; \bar{U} \quad\triangleq\quad (\forall U \in \bar{U})(I\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{type\text{-}extends}\; U)$$

Judgment $C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{impl}\; U$ expresses that: (1) class C implements an interface U, (2) methods and predicates declared in interface U are implemented in C, and (3) methods and predicates implemented in C are subtypes of the corresponding methods and predicates declared in U:

$$C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{impl}\; U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \;\Rightarrow\; \mathsf{mtype}(m, C\langle\bar{\alpha}\rangle) \neq \mathsf{undef}) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \;\Rightarrow\; \mathsf{ptype}(P, C\langle\bar{\alpha}\rangle) \neq \mathsf{undef}) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \;\Rightarrow \\ \qquad\qquad \bar{\alpha} : \bar{T} \vdash \mathsf{mtype}(m, C\langle\bar{\alpha}\rangle) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \;\Rightarrow\; \mathsf{ptype}(P, C\langle\bar{\alpha}\rangle) <: pt) \end{cases}$$

$$C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{impl}\; \bar{U} \quad\triangleq\quad (\forall U \in \bar{U})(C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{impl}\; U)$$

Verified Interfaces and Classes. In this paragraph, we define what are verified interfaces and classes. Later, when we verify a user-provided program, we will assume that the class table (i.e., a set of interfaces and classes) is verified.

Well-formed Predicate Types, $\Gamma \vdash pt : \diamond$, and Method Types, $\Gamma \vdash mt : \diamond$:

(Pred Type)
$$\frac{\Gamma \vdash \bar{T} : \diamond}{\Gamma \vdash \texttt{pred}\; P\langle\bar{T}\,\bar{\alpha}\rangle : \diamond}$$

(Mth Type)
$$\frac{\Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V} \vdash \bar{T}, F, U, \bar{V} : \diamond \quad \Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V}, \texttt{result} : U \vdash G : \diamond}{\Gamma \vdash \langle\bar{T}\,\bar{\alpha}\rangle\,\texttt{requires}\; F; \texttt{ensures}\; G;\; U\; m(\bar{V}\,\bar{\imath}) : \diamond}$$

Verified Interfaces, $int : \diamond$:

(Ax)
$$\frac{\Gamma \vdash F : \diamond}{\Gamma \vdash \texttt{axiom}\; F : \diamond}$$

(Int)
$$\frac{I\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{type\text{-}extends}\; \bar{U} \quad \texttt{init} \notin \mathsf{dom}(pt\texttt{*}) \qquad \bar{\alpha} : \bar{T} \vdash \bar{T}, \bar{U}, pt\texttt{*} : \diamond \quad \bar{\alpha} : \bar{T}, \texttt{this} : I\langle\bar{\alpha}\rangle \vdash ax, mt\texttt{*} : \diamond}{\texttt{interface}\; I\langle\bar{T}\,\bar{\alpha}\rangle\,\texttt{ext}\;\bar{U}\;\{pt\texttt{*}\;ax\texttt{*}\;mt\texttt{*}\} : \diamond}$$

Below, we write $\mathsf{cfv}(c)$ for the set of variables that occur freely in an object creation command in c. Formally:

$$\mathsf{cfv}(c) = \{\alpha \in \mathsf{fv}(c) \mid \alpha \text{ occurs in an object creation command } \ell = \texttt{new}\; C\langle\bar{\pi}\rangle\}$$

Rule (Cls) below is the main judgment for verifying classes. Premises $C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{ext}\; U$ and $C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{impl}\; \bar{V}$ enforce class C to be sane. Premise $C\langle\bar{T}\,\bar{\alpha}\rangle \;\mathsf{sound}$ enforces C's axioms

to be sound. Premise $\bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>} \vdash fd\texttt{*}, ax\texttt{*}, md\texttt{*} : \diamond$ enforces C's methods ($md\texttt{*}$) to be verified.

Rule (Mth) below verifies methods. In this rule, we prohibit object creation commands to contain logical method parameters because our operational semantics does not keep track of logical method parameters (while it does keep track of class parameters).

Verified Classes, $cl : \diamond$**:**

$$\frac{\begin{array}{c}\text{(Cls)}\quad C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{ext}\ U \quad C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{impl}\ \bar{V} \quad C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathsf{sound} \quad \texttt{init} \notin \mathsf{dom}(pd\texttt{*}) \\ \bar{\alpha} : \bar{T} \vdash \bar{T}, U, \bar{V} : \diamond \quad \bar{\alpha} : \bar{T} \vdash pd\texttt{*} : \diamond\ \mathsf{in}\ C\texttt{<}\bar{\alpha}\texttt{>} \quad \bar{\alpha} : \bar{T}, \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>} \vdash fd\texttt{*}, ax\texttt{*}, md\texttt{*} : \diamond\end{array}}{\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*}\texttt{\}} : \diamond}$$

$$\frac{\text{(Fld)}\quad \Gamma \vdash T : \diamond}{\Gamma \vdash T\ f : \diamond} \qquad \frac{\text{(Pred)}\quad \Gamma \vdash \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} : \diamond \quad \Gamma, \texttt{this} : U, \bar{\alpha} : \bar{T} \vdash F : \diamond}{\Gamma \vdash \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F : \diamond\ \mathsf{in}\ U}$$

$$\frac{\begin{array}{c}\text{(Mth)}\quad \Gamma \vdash \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} : \diamond \quad \mathsf{cfv}(c) \cap \bar{\alpha} = \emptyset \\ \Gamma' = \Gamma, \bar{\alpha} : \bar{T}, \bar{\imath} : \bar{V} \quad \Gamma'\texttt{;}\ \texttt{this} \vdash \{F\texttt{*}\ \texttt{this} \neq \texttt{null}\}c : U\{\texttt{(ex}\ U\ \texttt{result)}\texttt{(}G\texttt{)}\}\end{array}}{\Gamma \vdash \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}} : \diamond}$$

3.5 Verified Programs

We now have all the machinery to define what is a verified program. To do so, we extend our verification rules to runtime states. Of course, the extended rules are never used in verification, but instead define a global state invariant, $st : \diamond$, that is preserved by the small-step rules of our operational semantics.

Our forcing relation $\models$ from Section 3.2.5 assumes formulas without logical variables: we deal with those by substitution, ranged over by $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$. We let $(\Gamma \vdash \sigma : \Gamma')$ whenever $\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma')$ and $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$ for all α in $\mathsf{dom}(\sigma)$.

Now, we extend the Hoare triple judgment to states:

$$\frac{\begin{array}{c}\Gamma \vdash \sigma : \Gamma' \qquad \mathsf{dom}(\Gamma') \cap \mathsf{cfv}(c) = \emptyset \qquad \Gamma, \Gamma' \vdash s : \diamond \\ \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \qquad \Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{G\}\end{array}}{\langle h, c, s\rangle : \diamond}\ \text{(State)}$$

The rule for states ensures that there exists a resource $\mathcal{R}$ to satisfy the state's command. The object identifier r in the Hoare triple (last premise) is the current receiver, needed to determine the scope of abstract predicates. Rule (State) enforces the current command to be verified with precondition F and postcondition G. No condition is required on F and G, but note that, by the semantics of Hoare triples, F represents the state's allocated memory before executing c: if c is not a top level program (i.e., some memory should be allocated for c to execute correctly), choosing a trivial F such as `true` is incorrect. Similarly, G represents the state's memory after executing c.

The judgment $(ct : \diamond)$ is the top-level judgment of our source code verification system, to be read as "class table ct is verified". We have shown the following theorem:

Theorem 3 (Preservation)**.** *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$, *then* $(st' : \diamond)$.

Proof. See Section 6.1.2. □

The proof of Theorem 3 means soundness of all rules presented so far. In particular, in this proof, we show that the Hoare rules from Section 3.3 are sound.

From the preservation theorem, we can draw two corollaries: verified programs never dereference `null` and verified programs satisfy partial correctness. To formally state these theorems, we say that a class table ct together with a "main" program c are sound (written $(ct, c) : \diamond$) iff ($ct : \diamond$ and $\texttt{null}; \emptyset \vdash \{\texttt{true}\}c : \texttt{void}\{\texttt{true}\}$). In the latter judgment, $\emptyset$ represents that the type environment is initially empty, `null` represents that the receiver is initially vacuous, and `true` represents that the top level program has `true` both as a precondition and as a postcondition. `true` is a correct precondition for top level programs (Java's `main`), because when a top level program starts to execute, the heap is initially empty.

We can now state the first corollary (no `null` dereference) of the preservation theorem. A head command hc is called a *null error* iff $hc = (\ell\,\texttt{=}\,\texttt{null}.f)$ or $hc = (\texttt{null}.f\,\texttt{=}\,v)$ or $hc = (\ell\,\texttt{=}\,\texttt{null}.m\texttt{<}\bar{\pi}\texttt{>(}\bar{v}\texttt{)})$ for some $\ell, f, v, m, \bar{\pi}, \bar{v}$.

Theorem 4 (Verified Programs are Null Error Free)**.** *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, hc; c', s\rangle$*, then* hc *is not a null error.*

Proof. See Section 6.1.3. □

To state the second corollary (partial correctness) of the preservation theorem, we extend head commands with *specification commands*. Specification commands sc are used by the proof system, but are ignored at runtime. The specification command `assert(`F`)` makes the proof system check that F holds at this program point:

$$\begin{array}{rcl} hc \in \mathsf{HeadCmd} & ::= & \ldots \mid sc \mid \ldots \\ sc \in \mathsf{SpecCmd} & ::= & \texttt{assert(}F\texttt{)} \end{array}$$

We update Section 2.2's operational semantics to deal with specification commands. Operationally, specification commands are no-ops:

State Reductions, $st \rightarrow_{ct} st'$**:**

(Red No Op)
$$\ldots \quad \langle h,\ sc;\ c, s\rangle \rightarrow \langle h,\ c, s\rangle \quad \ldots$$

Now, we can state the partial correctness theorem. It expresses that if a verified program contains a specification command `assert(`F`)`, then F holds whenever the assertion is reached at runtime:

Theorem 5 (Partial Correctness)**.**
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, \texttt{assert(}F\texttt{)}; c, s\rangle$*, then* $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$ *for some* $\Gamma, \mathcal{E} = \mathcal{F}_{ct}(\mathcal{E}), \mathcal{P}$ *and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

Proof. See Section 6.1.3. □

3.6 Examples of Reasoning

3.6.1 Example: Roster

This example consists of a linked list implementation of a class roster that collects student identifiers and associates them with grades. We design the roster class so that multiple threads can concurrently read a roster (this can be expressed in contracts, even though we do not have multiple threads yet). Moreover, when a thread updates the grades we allow other threads to concurrently read the student identifiers.

Objects of type `Roster` have two predicates with two permission parameters each:

`ids_and_links<p,q>`	predicate including student ids and links between the student entries `p` is the permission for the student ids `q` is the permission for the links
`grades_and_links<p,q>`	predicate including grades and links between the student entries `p` is the permission for the grades `q` is the permission for the links

As the two predicates overlap on the links, the two predicates do not give the right to change the links. Predicate `ids_and_links` allows to change ids if `p` is `1` while predicate `grades_and_links` allows to change grades if `p` is `1`. To change the links, the two predicates *together* are needed. This is expressed by predicate `state` in `Roster`'s implementation below.

```
class Roster{
 int id; int grade; Roster next;
 pred ids_and_links<perm p, perm q> =
  Perm(id,p) *
  (ex Roster x)(PointsTo(next,q,x) * x.ids_and_links<p,q>);
 pred grades_and_links<perm p,perm q> =
  Perm(grade,p) *
  (ex RosterImpl x)(PointsTo(next,q,x) * x.grades_and_links<p,q>);
 public pred state<perm p> = ids_and_links@Roster<p,p/2> *
                             grades_and_links@Roster<p,p/2>;
 requires init * n.state<1>; ensures state@Roster<1>;
 void init(int i, int g, Roster n) {
  this.id = i; this.grade = g; this.next = n;
 }
 requires grades_and_links<1,p> * ids_and_links<q,r>;
 ensures grades_and_links<1,p> * ids_and_links<q,r>;
 void updateGrade(int id, int grade) {
  if (this.id == id) { this.grade = grade; }
  else if (next != null) { next.updateGrade(id,grade); }
 }
 requires ids_and_links<p,q>; ensures ids_and_links<p,q>;
 bool contains(int id) {
  bool b = this.id==id; if(!b && next!=null){ b=next.contains(id); }; b
 }
```

```
}
```

Here are informal interpretations of the method contracts:

`updateGrade(id,grade):` Requires write access to the grades and read access to the student ids and the links. (We omit quantifiers over the logical variables `p`, `q`, `r` because they can be inferred.)

`contains(id):` Requires read access to the student ids and the links. (We omit quantifiers over the logical variables `p`, `q` because they can be inferred.)

Our implementation also contains an `init`-method:

`init(id,grade,next):` Plays the role of a constructor. Requires write access to the fields of `this` (by precondition `init`) and write access to the state of `next` (by precondition `n.state<1>`). Ensures write access to the roster (by postcondition `state@RosterImpl<1>`).

Note that method `init`'s postcondition refers to the `Roster` class. This might look like breaking the abstraction provided by subtyping. However, because method `init` is meant to be called right after object creation (using `new Roster`), `init`'s postcondition can be converted into a form that does not mention the `Roster` class. Given an object o, after calling o `= new Roster` and o`.init()`, the caller knows that `Roster` is o's dynamic class (recall that (New)'s postcondition includes an `classof` predicate) and can therefore convert the access ticket o`.state@Roster<1>` to o`.state<1>` (using axiom (Known Type)).

Below, we verify method `init`. We do not give proof outlines for the two other methods, because they are less interesting.

```
 { init * n.state<1> }
 ((Dynamic Type) axiom)
 { init * init@Roster ispartof init * n.state<1> }
 (Unfolding ispartof definition)
 { init * init -* (init@Roster -* (init@Roster -* init)) * n.state<1> }
 (Modus ponens)
 { init@Roster * (init@Roster -* init) * n.state<1> }
 (Weakening)
 { init@Roster * n.state<1> }
 ((Open/Close) axiom)
 { Perm(this.id,1) * Perm(this.grade,1) * Perm(this.next,1) * n.state<1> }
this.id = id;
 { Perm(this.id,1) * Perm(this.grade,1) * Perm(this.next,1) * n.state<1> }
this.grade = grade;
 { Perm(this.id,1) * Perm(this.grade,1) * Perm(this.next,1) * n.state<1> }
this.next = n;
 { Perm(this.id,1) * Perm(this.grade,1) * PointsTo(this.next,1,n) *
   n.state<1> }
 ((Open/Close) axiom)
 { Perm(this.id,1) * Perm(this.grade,1) * PointsTo(this.next,1,n) *
   n.ids_and_links<1,1/2> * n.grades_and_links<1,1/2> }
 ((Split/Merge) axiom)
 { Perm(this.id,1) * PointsTo(this.next,1/2,n) * n.ids_and_links<1,1/2> *
```

```
   Perm(this.grade,1) * PointsTo(this.next,1/2,n) *
   n.grades_and_links<1,1/2> }
 ((Open/Close) axiom)
 { state@Roster<1> }
```

Because `init` mimics a constructors, we have to allow clients to call methods `updateGrade` and `contains` after `init` returned. To do this, we give clients the ability to switch between the `state` predicate (`init`'s postcondition) and the `grades_and_links` and `ids_and_links` predicates (that appear in `updateGrade` and `contains`'s contracts). For this, we use the `public` modifier (as indicated in `Roster`'s implementation) before predicate `state`'s definition. As explained in Section 3.2.1, the `public` modifier exports predicate `state`'s definition in class `Roster` to clients.

3.6.2 Example: Iterator

In this example, we show how we can precisely specify the usage protocol of Java's `Iterator` interface. We do not give an implementation (which can be found in our earlier technical report [54]) but show how value-parameterized classes and class axioms are crucial to achieve our goal.

Often one wants to constrain object clients to adhere to certain usage protocols. Usage protocols can, for instance, be specified in typestate systems [40] or; using ghost fields, by general purpose specification languages [91]; or by dedicated specifications (as in Cheon et al.'s work [32] and in Chapter 7). A limitation of these techniques is that state transitions must always be associated with method calls. This is sometimes not sufficient. Consider for instance Java's `Iterator` interface (enriched with an `init` method because our model language does not have constructors).

```
interface Iterator{
  void init(Collection c);
  boolean hasNext();
  Object next();
  void remove();
}
```

If iterators are used in an undisciplined way, there is the danger of unwanted concurrent modification of the underlying collection (both of the collection elements and the collection itself). Moreover, in concurrent programs bad iterator usage can result in data races. It is therefore important that `Iterator` clients adhere to a usage discipline. The following simple discipline would be safe for an iterator without `remove`: *retrieve the next collection element; then access the element; then trade the element access right for the right to retrieve the next element; and so on.* Although such a discipline is simple and makes sense, it cannot be specified by existing typestate systems and it would be very clumsy to specify it with classical specification languages because they can only express transitions associated with method calls.

We have designed a usage protocol for the full `Iterator` interface with `remove`. Its state machine is shown in Figure 3.2. The dashed arrows are the ones that are not

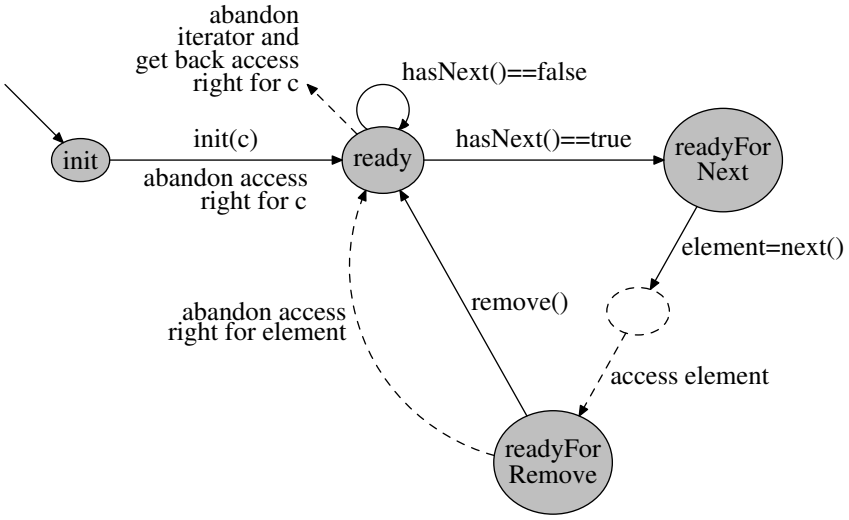

Figure 3.2: Usage protocol of the `Iterator` interface

associated with method calls, and are hard to capture with existing object-oriented specification systems. Note in particular, that according to this protocol an `Iterator` client can keep the access right for a collection element that he has removed. This protocol can be expressed quite straightforwardly by a separation logic contract (making heavy use of linear implication).

```
interface Iterator<perm p, Collection iteratee>{
   pred ready;                              // prestate for iteration cycle
   pred readyForNext;                       // prestate for next()
   pred readyForRemove<Object element>;  // prestate for remove()
   axiom ready -* iteratee.state<p>;     // stop iterating
   requires init * c.state<p> * c==iteratee;
   ensures ready;
   void init(Collection c);
   requires ready;
   ensures (result -* readyForNext) & (!result -* ready);
   boolean hasNext();
   requires readyForNext;
   ensures result.state<p> * readyForRemove<result> *
         ((result.state<p> * readyForRemove<result>) -* ready);
   Object next();
   requires readyForRemove<_> * p==1;
   ensures ready;
   void remove();
}
```

The interface has two parameters: firstly, a permission `p` and, secondly, the iteratee. If the permission parameter is instantiated by a fraction $\texttt{p} \leq 1$, one obtains a read-only iterator, otherwise a read-write iterator. The states are represented by three abstract predicates. The class axiom expresses that whenever the client is in the `ready` state, he has the option to abandon the iterator for good and get the access right for the iteratee back. The precondition of `init()` consumes a fraction `p` of the access right for the iteratee and puts the iterator in the `ready` state. The `init` predicate in `init()`'s precondition is a special abstract predicate that every object enters right after object creation and that grants access to all of the object's fields (recall that our model language does not have constructors). The most interesting part of the `Iterator` contract is `next()`'s postcondition. It grants access to the collection element that got returned, represented by the special `result` variable. Furthermore, it grants permission to remove this element. However, by the precondition of `remove`, this permission can only be used if the class parameter `p` is 1, i.e., the iterator is read-write. Finally, `next()`'s postcondition grants the right to trade the tickets `result.state<p>` and `readyForRemove<result>` for the `ready` state.

We have implemented this interface for a doubly linked list implementation of the `Collection` interface (it is detailed in our earlier work [54]). This implementation revealed to be time consuming: finding the correct definitions of predicates `ready`, `readyForNext`, `readyForRemove`, and the predicates of the doubly linked list was an intricate task. Intuitively, the difficulty lies in the nit-picking reasoning done at the level of fields of the doubly linked list. The difficulties encountered when writing this implementation spawned the idea underlying the work on class protocols shown in Chapter 7.

Work Related with the `Iterator` *Example.* Recently, iterators have served as a challenging case study for several verification systems, namely, separation logic [89], higher-order separation logic [74], a linear typestate system [17, 18], and a linear type-and-effect system [26].

Parkinson [89] uses iterators as an example. He supports simultaneous read-only iterators through counting permissions [21], rather than fractional permissions. Unlike us, he considers iterators over shallow collections, i.e., iterators that do not own their underlying collections. His iterator interface does not have a `remove()` method, which is particularly interesting for deep collections, because it is important that the collection passes the access permission for removed elements to the remover. Parkinson's proof of his iterator implementation differs from ours: he uses a lemma that is proven inductively, whereas our proof is based on the natural deduction rules for our fragment of separation logic without an induction rule.

Krishnaswami [74] presents a protocol for iterators over linked lists using *higher-order* separation logic. His collections are shallow and his iterators are read-only. His protocol allows multiple iterators over a collection and enforces that all active iterators are abandoned, once a new element is added to the collection. Technically, he achieves this by parameterizing a higher-order predicate by a first-order predicate that represents the state of a collection. Iterators that are created when the underlying collection is in a certain state P may only be used as long as the collection is still in state P. The `add()` method does not preserve the collection state and hence invalidates all existing iterators. This is an elegant solution that makes use of the power of higher-order predicates.

Bierhoff and Aldrich [18] present a linear typestate system based on a fragment of

linear logic. It uses linear implication as a separator between pre- and postconditions (but not as a logical connective). Bierhoff and Aldrich use iterators as a case study for their system [18, 17]. They support concurrent read-only iterators through fractional permissions. Their protocols do not support iterators over deep collections with *mutable* collection elements, although [17] supports *read-only* access to collection elements that get returned by `next()`. Our protocol cannot be represented in their system because their specification language lacks linear implication (needed to represent the dashed `readyForRemove`-to-`ready` transition). Of course, they could add linear implication to their language. They associate our second dashed transition (the one that terminates an iteration) with the iterator's `finalize()` method, and assume that a checker would employ program analysis techniques to apply the `finalize()`-contract without explicitly calling `finalize()`. In practice, this has the same effect as our axiom. Neither of Bierhoff and Aldrich's works [18, 17] presents an iterator implementation, or a mapping of iterator state predicates to concrete definitions. To verify iterator implementations, related verification systems employ recursive predicates and either induction [89] or introduction and elimination rules for linear implication (this thesis) or both [74]. Recursive predicates, induction and linear implication are not supported by Bierhoff and Aldrich's works, and it is thus possible that their works are not expressive enough to verify iterator implementations. Of course, it is likely that Bierhoff and Aldrich have deliberately avoided such features (especially induction) because their system is designed as a lightweight typestate system, rather than a full-blown program logic.

Boyland, Retert and Zhao [26] informally explain how to apply their linear type and effect system (an extension of [25] with fractional permissions) to specify and verify iterator protocols. Their system facilitates concurrent read-only iterators through fractional permissions. The paper does not address iterators over deep collections. In contrast to Boyland et al.'s [26] `Iterator` interface, our `Iterator` interface is parameterized by the underlying collection. As a result, in our system client methods and classes sometimes need an auxiliary parameter (in angle brackets). Like us, Boyland et al. use linear implication to represent the state transition that finalizes an iterator. They represent this linear implication as an effect on the `iterator()` method, whereas we choose to represent it as a class axiom. Their linear implication operator (called "scepter") has a different semantics than separation logic's magic wand, which we use.

3.7 Related Work and Conclusion

Related Work. The closest related work is Parkinson's thesis [88] which provides verification for Java-like programs specified with separation logic. We reuse most of Parkinson's ideas but differ on some points. We have a more restrictive rule for predicate extensions in subclasses and for method subtyping. On the upside, we do not have to reverify inherited methods; on the downside, our notion of subtyping is more restrictive. Recently [33, 90], new approaches show how to get a notion of subtyping that is more powerful both than Parkinson's early work and our work. Another difference with Parkinson's work is that we provide value-parameterized classes and methods, class axioms, and public modifiers. These features are crucial to handle advanced examples such as Java's `Iterator`.

Verifiers for object-oriented programs include ESC/Java2 [35] for Java and Boogie for C# [6]. On one hand, ESC/Java2 offers many facilities that we do not feature (arrays, ghost fields, model classes and methods, etc.). On the other hand, ESC/Java2 uses tradi-

tional Hoare logic. As outlined in the thesis's introduction, this has the consequence that alias reasoning becomes difficult. Another difference is that ESC/Java2 uses a weakest precondition calculus to generate proof obligations, whereas implementation of our verification would be done with a symbolic execution algorithm (à la smallfoot [15]). On the level of specifications, Boogie is similar to ESC/Java2. It includes, however, a permission model based on invariants to help reasoning about aliasing. One advantage is that reasoning can still be done in first order logic, for which very good solvers exist. A disadvantage is that the permission model is fixed (it is *not* supplied by programmers) and it can be too restrictive for intricate classes. Contrary to us, both ESC/Java2 and Boogie use an intermediate language to generate proof obligations.

Dynamic frames [95, 94] is another approach to verify programs. In this approach, programmers use *pure* methods (methods that do no write to the heap) and ghost fields to specify sets of locations that are accessed by normal methods. Specifying the sets of locations accessed by methods mimics separation logic's permissions. Like ESC/Java2 and Boogie, program verifiers using dynamic frames generate proof obligations in first order logic.

Conclusion. We presented how to specify and verify programs written in Chapter 2's language with separation logic. To do this, we presented a set of Hoare rules that we have shown sound. The key contributions of this chapter are: (1) for expressiveness, we provide parameterized classes and methods, (2) we provide a subtyping relation that avoid reverification of inherited methods, and (3) to export useful facts to clients, we use axioms and `public` modifiers.

In the next chapter, we extend our model language and our verification system to deal with multithreaded programs that use fork and join as concurrency primitives.

Chapter 4

Separation Logic for fork/join

In this chapter, we extend Chapter 2's language with multithreading. Multithreading is achieved with fork and join primitives à la Java. Further, we extend Chapter 3's assertion language and verification rules to deal with fork and join primitives.

This chapter is structured as follows: in Section 4.1 we informally present what fork and join primitives are, in Section 4.2 we show how to modify Section 2.1's language to model fork and join, in Section 4.3 we describe how to modify Section 3.2's assertion language and semantics to handle fork and join, in Section 4.4 we present the verification rules for fork and join, and in Section 4.5 we show what are verified programs. We present two examples of verified programs in Section 4.6: a simple example of a parallel computation of the Fibonacci sequence and an advanced example of replication of a database to mirrors. Finally, we discuss related work and conclude in Section 4.7.

Note: The work from this chapter has been done in collaboration with Christian Haack. It is published in the International Conference on Algebraic Methodology and Software Technology (AMAST 2008) [53]. Compared to [53], the treatment of `run`'s contract is simpler and inheritance is handled in a better way.

Conventions: In formal material, we use grey background to highlight what are the changes compared with previous chapters. As in the previous chapter, we use dots "..." to indicate that some material defined previously is being extended.

4.1 Fork/join: Background

As sketched in the thesis's introduction, multithreaded programs are becoming ubiquitous, because of (1) the high demand on software and of (2) the recent trend to increase the number of processors on hardware.

Fork and join primitives are the main mechanisms for concurrency in a number of programming languages including Java, C♯ (in these languages, fork is called *start*), python, C++ etc. Fork starts a new *thread* i.e., it starts execution of code in parallel with the code already running. Fork permits *dynamic thread creation*, because the number of threads that is created during a program's run is not statically determined: threads are started when fork commands are executed. This is dynamic, because execution of fork commands can depend on program variables which also depend on uncontrolled input.

Join's purpose is to wait for a thread to terminate: calling join on a thread blocks until the thread has terminated, and then returns. An example usage of join is to wait for some IO operation (like reading to the disk) to finish before inspecting the data obtained. This can be performed by forking a new thread to perform the IO operation and by later joining this thread.

In this chapter, we extend Chapter 3 with fork and join primitives. This includes extending our verification system with rules for these primitives and extending our assertion language to model join's behavior (nothing special will be needed for fork). As sketched in the thesis's introduction, we will see that separation logic is suited to reason about concurrent programs, because it encompasses permissions to control access to the heap. Because two write permissions (or a write permission and a read permission) to the same cell cannot coexist, programs verified in our system are data race free by construction.

4.2 A Java-like Language with fork/join

We show how to extend the syntax and the semantics of Section 2.1's language with fork and join primitives.

Syntax. From now on, we assume that class tables always contain the declaration of class `Thread`. Class `Thread` includes methods `fork`, `join`, and `run`:

```
class Thread extends Object{
 final void fork();
 final void join();
 final void run() { null }
}
```

The methods `fork` and `join` are not implemented in Java. Instead, they are implemented natively and their behavior is specified in the operational semantics as follows:

- o.`fork()` creates a new thread, whose thread identifier is o, and executes o.`run()` in this thread. Method `fork` should not be called more than once on o: any subsequent call results in blocking of the calling thread.
- o.`join()` blocks until thread o has terminated.

The `run`-method is meant to be overridden while methods `join` and `fork` cannot be overridden (as indicated by the `final` modifiers). In Java, `fork` and `join` are not final. In combination with super calls, this is useful for custom `Thread` classes. However, we leave the study of overrideable `fork` and `join` method together with super calls as future work, and stick to our setting for simplicity.

Runtime Structures. In Section 2.2, our operational semantics $\rightarrow_{ct}$ was defined to operate on states consisting of a heap, a command, and a stack. To account for multiple threads, states are modified to contain a heap and a *thread pool*. A thread pool maps object identifiers (representing `Thread` objects) to threads. Threads consist of a thread-local stack s and a continuation c. For better readability, we use syntax-like notation and write "s in c" for threads $t = (s, c)$, and "o_1 is t_1 $\mid$ $\cdots$ $\mid$ o_n is t_n" for thread pools

$ts = \{o_1 \mapsto t_1, \ldots, o_n \mapsto t_n\}$:

$$\begin{array}{rcll} t \in \mathsf{Thread} & = & \mathsf{Stack} \times \mathsf{Cmd} & ::= \quad s \text{ in } c \\ ts \in \mathsf{ThreadPool} & = & \mathsf{ObjId} \rightharpoonup \mathsf{Thread} & ::= \quad o_1 \text{ is } t_1 \mid \cdots \mid o_n \text{ is } t_n \\ st \in \mathsf{State} & = & \mathsf{Heap} \times \mathsf{ThreadPool} & \end{array}$$

Initialization. We modify Section 2.2's definition of the initial state of a program. Below, main is some distinguished object id for the main thread. The main thread has an empty set of fields (hence the first $\emptyset$), and its stack is initially empty (hence the second $\emptyset$):

$$\mathsf{init}(c) = \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\},\ \mathsf{main} \text{ is } (\emptyset \text{ in } c)\rangle$$

The *operational semantics* defined in Section 2.2 is straightforwardly modified to deal with multiple threads. In each case, *one* thread proceeds, while the other threads remain untouched. In addition, to model `fork` and `join`, we change the reduction step (Red Call) to model that it does not apply to `fork` and `join`. Instead, `fork` and `join` are modeled by two new reductions steps ((Red Fork) and (Red Join)):

State Reductions, $st \rightarrow_{ct} st'$**:**

...

(Red Call) $\quad m \notin \{\texttt{fork}, \texttt{join}\}$
$h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\iota_0; \bar{\iota}).c_m \quad c' = c_m[o/\iota_0, \bar{v}/\bar{\iota}]$
$\langle h,\ ts \mid p \text{ is } (s \text{ in } \ell = o.m(\bar{v});\ c)\rangle \rightarrow \langle h,\ ts \mid p \text{ is } (s \text{ in } \ell \leftarrow c';\ c)\rangle$

(Red Fork) $\quad h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad o \notin (\mathsf{dom}(ts) \cup \{p\}) \quad \mathsf{mbody}(\texttt{run}, C\texttt{<}\bar{\pi}\texttt{>}) = (\iota).c_r \quad c_o = c_r[o/\iota]$
$\langle h,\ ts \mid p \text{ is } (s \text{ in } \ell = o.\texttt{fork}();\ c)\rangle \rightarrow \langle h,\ ts \mid p \text{ is } (s \text{ in } \ell = \texttt{null};\ c) \mid o \text{ is } (\emptyset \text{ in } c_o)\rangle$

(Red Join)
$\langle h,\ ts \mid p \text{ is } (s \text{ in } \ell = o.\texttt{join}();\ c) \mid o \text{ is } (s' \text{ in } v)\rangle \rightarrow \langle h,\ ts \mid p \text{ is } (s \text{ in } \ell = \texttt{null};\ c) \mid o \text{ is } (s' \text{ in } v)\rangle$

...

Remarks.

- In (Red Fork), a new thread o is forked. Thread o's state consists of an empty stack $\emptyset$ and command c_0. Command c_o is the body of method `run` in o's dynamic type where the formal receiver `this` and the formal class parameters have been substituted by the actual receiver and the actual class parameters.
- In (Red Join), thread p joins the terminated thread o. Our rule ensures that thread o is terminated because its command consists of a single value v.

4.3 Separation Logic for fork/join

In this section, we extend our assertion language to deal with `fork` and `join` primitives. We introduce (1) a `Join` predicate that controls how threads access postconditions of terminated threads and (2) `groups` which are a restricted class of predicates. Groups are needed to allow multiple threads to join a terminated thread.

4.3.1 The `Join` predicate

To model `join`'s behavior, we add a new formula to the assertion language defined in Section 3.2. This formula will be used (see Section 4.4) to govern exchange of permissions from terminated threads to alive threads:

$$F \ ::= \ \ldots \mid \texttt{Join}(e, \pi) \mid \ldots$$

The intuitive meaning of $\texttt{Join}(e, \pi)$ is as follows: this formula allows to pickup fraction π (recall that permissions are fractions in $(0, 1]$) of thread e's postcondition after e terminated. As a specific case, if π is 1, the thread in which the `Join` predicate appears can pickup thread e's entire postcondition when e terminates.

We extend Section 3.2.2's judgment for well-typed formulas as follows:

Well-typed Formulas, $\Gamma \vdash F : \diamond$:

$$\ldots \quad \frac{\Gamma \vdash e : \texttt{Thread} \quad \Gamma \vdash \pi : \texttt{perm}}{\Gamma \vdash \texttt{Join}(e, \pi) : \diamond} \ (\text{Form Join}) \quad \ldots$$

The `Join` predicate is emitted when new threads are created. To model this, we redefine the `init` predicate (recall that `init` appears in (New)'s postcondition) for subclasses of `Thread` and for other classes. We do that by (1) adding the following clause to the definition of predicate lookup:

$$\mathsf{plkup}(\texttt{init}, \texttt{Thread}) = \texttt{pred init = Join(this, 1) ext Object}$$

and (2) adding $C \neq \texttt{Thread}$ as a premise to (Plkup `init`). Intuitively, when an object o inheriting from `Thread` is created, a $\texttt{Join}(o, 1)$ ticket is issued.

Resources. To express the semantics of the `Join` predicate, we need to change our definition of models (resources). Recall that, in Section 3.2.3, resources were couples of a heap and a permission table of type $\mathsf{ObjId} \times \mathsf{FieldId} \to [0, 1]$. We modify permission tables so that they have type $\mathsf{ObjId} \times (\mathsf{FieldId} \times \{\texttt{join}\}) \to [0, 1]$. The additional element in the domain of permission tables keeps track of how much a thread can pick up of another thread's postcondition. In other words, in the previous chapter, the meaning of resources was to regulate access (write access, readonly access, or no access) to field of objects; now resources still have this meaning, *plus* they regulate access of threads to other threads's postconditions. Obviously, we forbid `join` to be a valid field identifier to avoid confusion between `join`'s special meaning and programmer-declared fields.

In addition, we add an element to resources; they become triples of a heap, a permission table, and a *join table* $\mathcal{J} \in \mathsf{ObjId} \to [0, 1]$. Intuitively, for a thread o, $\mathcal{J}(o)$ keeps track of how much of o's postcondition has been picked up by other threads: when a thread gets joined, its entry in $\mathcal{J}$ drops. The compatibility and joining operations on join tables are defined as follows:

$$\mathcal{J} \# \mathcal{J}' \text{ iff } \mathcal{J} = \mathcal{J}' \qquad \mathcal{J} * \mathcal{J}' \triangleq \mathcal{J}$$

Because # is equality, join tables are "global": in the preservation proof, all resources will have the same join table[1].

Now, we require resources to satisfy these axioms (in addition to Section 3.2.3's axioms):

(a) For all $o \notin \mathsf{dom}(h)$ and all f (including `join`), $\mathcal{P}(o, f) = 0$ and $\mathcal{J}(o) = 1$.
(b) $\lambda\, o\, .\, \mathcal{P}(o, \texttt{join}) \leq \mathcal{J}$.

Axiom (a) ensures that all objects that are not yet allocated have minimal permissions. This is needed to prove soundness of the verification rule for allocating new objects. Axiom (b) is a technical condition needed to prove soundness of the verification rule for joining threads.

As usual, we define a projection operator:

$$(h, \mathcal{P}, \mathcal{J})_{\mathsf{join}} \triangleq \mathcal{J}$$

Predicate Environments. For technical reasons, we need to update the definition of predicate environments given in Section 3.2.4 (see page 27). Specifically, we need predicate environments to satisfy the following additional axiom:

(b) If $(\bar\pi, (h, \mathcal{P}, \mathcal{J}), r, \bar\pi'), (\bar\pi, (h, \mathcal{P}, \mathcal{J}'), r, \bar\pi') \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathcal{P}(o, \texttt{join}) \leq x \leq \mathcal{J}(o)$, and $\mathcal{J}' = \mathcal{J}[o \mapsto x]$, then:
$$\mathcal{E}(\kappa)(\bar\pi, (h, \mathcal{P}, \mathcal{J}), r, \bar\pi') \leq \mathcal{E}(\kappa)(\bar\pi, (h, \mathcal{P}, \mathcal{J}'), r, \bar\pi')$$

Axiom (b) is used to update the global join table, because, when a thread is joined, its corresponding entry drops in all join tables.

Semantics. The semantics of the `Join` predicate is as follows:

$$\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \quad \models \quad \texttt{Join}(e, \pi) \text{ iff } [\![e]\!]^h_s = o \text{ and } [\![\pi]\!] \leq \mathcal{P}(o, \texttt{join})$$

Axiom. In analogy with the `PointsTo` predicate, we have a split/merge axiom for the `Join` predicate:

$$\Gamma; v \vdash \texttt{Join}(e, \pi) \mathbin{*\!\!-\!\!*} (\texttt{Join}(e, \tfrac{\pi}{2}) * \texttt{Join}(e, \tfrac{\pi}{2})) \qquad \text{(Split/Merge Join)}$$

4.3.2 Groups

In addition to `pred`, we use a new keyword `group` to declare predicates that have special properties. *Groups* will be needed in the next section to express that multiple threads can join a terminated thread. Groups are predicates that satisfy an additional split/merge axiom. Formally, `group` desugars to a `pred`icate and an `axiom`:

$$\texttt{group}\ P\texttt{<}\bar T\ \bar x\texttt{>} = F \quad \triangleq \quad \begin{array}{l}\texttt{pred}\ P\texttt{<}\bar T\ \bar x\texttt{>} = F\texttt{;} \\ \texttt{axiom}\ P\texttt{<}\bar x\texttt{>} \mathbin{*\!\!-\!\!*} (P\texttt{<split}(\bar T, \bar x)\texttt{>} * P\texttt{<split}(\bar T, \bar x)\texttt{>})\end{array}$$

where `split` is extended to pairs of type and parameter, so that it splits parameters of type `perm` and leaves other parameters unchanged:

$$\texttt{split}(T, x) \triangleq \begin{cases} \texttt{split}(x) & \text{iff } T = \texttt{perm} \\ x & \text{otherwise} \end{cases}$$

[1] This suggests that join tables could be avoided all together in resources. It is unclear, however, if an alternative approach would be cleaner because rules (State), (Cons Pool), and (Thread) would need extra machinery.

The meaning of the axiom for groups is as follows: (1) splitting (reading `*-*` from left to right) P's parameters splits P predicates and (2) merging (reading `*-*` from right to left) P's parameters merges P predicates.

For the group axiom above to make sense, extending the list of parameters of groups (recall that we allow to extend the list of parameters of predicates) should be forbidden in subclasses [54, §G]. Otherwise, if a subclass extends a group by adding parameters to it, the axiom for group becomes unprovable. Intuitively, because missing parameters of predicates are existentially quantified, one cannot soundly merge two groups where parameters are missing (the missing parameters might be different for the two groups being merged). This restriction could be slightly weakened by allowing extension of the list of parameters with parameters that have *unique witnesses* i.e., such that missing parameters are uniquely defined. We do not use this more liberal criterion, however, because we do not need it any examples of this thesis.

4.4 Contracts for fork and join

Next, we extend the verification system defined in the previous chapter to include `fork` and `join` primitives. As we give contracts for these primitives in a class, we do not need to give new Hoare rules. Rules for `fork` and `join` will simply be instances of the rule for method call (Mth). Contracts for `fork` and `join` model how permissions to access the heap are exchanged between threads. Intuitively, newly created threads are given a part of the heap by their parent thread. Dually, when terminated threads are joined, parts of the heap they used is given to joining threads.

Class `Thread`. In Section 4.2, we introduced class `Thread` but did not give any specifications. Class `Thread` is specified as follows:

```
class Thread extends Object{
 pred  preFork = true;
 group postJoin<perm p> = true;
 requires preFork; ensures true;
 final void fork();
 requires Join(this,p); ensures postJoin<p>;
 final void join();
 final requires preFork; ensures postJoin<1>;
 void run() { null }
}
```

The contracts of `fork`, `join`, and `run` are tightly related: (1) `fork`'s precondition is similar to `run`'s precondition and (2) `run`'s postcondition includes predicate `postJoin<1>` while `join`'s postcondition is `postJoin<p>`. Point (1) models that when a thread is forked, its `run` method is executed: part of the parent thread's state is passed to the forked thread. Point (2) models that `join` returns after `run` terminated. Further, (2) represents that threads joining a thread might pick up a part of the joined thread's state. The fact that permission `p` appears both as an argument to `Join` and to `postJoin` (in `join`'s contract) models that joining threads pick up a part of the terminated thread's state which is *proportional* to `Join`'s argument. Because one `Join(`$o, 1$`)` predicate is issued

per thread o, our system enforces that threads joining o do not pick up more than thread o's postcondition.

For the behavior just described to be sound, we require that `postJoin` is a group. This requirement is required to prove soundness of `join`'s contract (see proof of Theorem 3 in Section 6.2.2 and Lemma 16). Intuitively, this requirement is needed because `join`'s postcondition (i.e., `postJoin`) is split among several threads. Hence, we have to make sure this splitting is sound.

Although method `run` is meant to be overridden, we require that method `run`'s contract cannot be modified in subclasses of `Thread` (as indicated by a `final` modifier). Enforcing `run`'s contract to be fixed allows to express that `join`'s postcondition is proportional to the second parameter of `Join`'s predicate in an easy way (because we can assume that `run`'s postcondition is always `postJoin<1>`). Fixing `run`'s contract in class `Thread` means that programmers have to specify `run` in `preFork` and `postJoin`'s definitions. In our examples (see Section 4.6 and Section 5.6), having `run`'s contract fixed proved to be convenient. However, it could possibly be inconvenient in more intricated examples. A large case study is needed to answer this issue accurately.

Because `run`'s contract is fixed, `run`'s contract cannot be parameterized by logical parameters. One could consider that this reduces expressiveness. But this is wrong, because logical parameters for `run` are unsound. As `run`'s pre and postconditions are interpreted in different threads, one cannot guarantee that logical parameters are instantiated in a similar manner between callers to `fork` and callers to `join`. Hence, logical parameters have to be forbidden for `run`.

We highlight that method `run` can also be called directly, without forking a new thread. Our system allows such behavior which is used in practice to flexibly control concurrency (cf Java's `Executors` [80]).

Alternative Solutions. Alternatively, we could allow arbitrary contracts for `run`, as we did in our previous work [54]. This solution, however, has the disadvantage that we need to introduce a new derived form for formulas (called *scalar multiplication*) at the level of method's contracts. With this thesis's solution, we can "hide" scalar multiplication "under the carpet" (see Section 4.5), i.e., we avoid scalar multiplication to spread in method contracts and in proofs of programs (even if we need it to *prove* our verification system sound). In addition, our earlier work breaks subtyping because, to start a thread, one has to know the static type of the thread considered one level (in the class hierarchy) below class `Thread`.

Yet another solution would be to combine (1) our approach of specifying `fork`, `join`, and `run` with predicates in class `Thread` and (2) to use scalar multiplication as a new *constructor* for formulas (i.e., not a derived form) to express that `run`'s postcondition can be split among joiner threads. This solution, however, requires a thorough study because having scalar multiplication as a new constructor for formulas may raise semantical issues (as studied by Boyland [24]).

4.5 Verified Programs

We need to update Section 3.5's rules to account for multiple threads. First, we craft rules for thread pools:

$$\frac{}{\mathcal{R} \vdash \emptyset : \diamond} \text{ (Empty Pool)} \qquad \frac{\mathcal{R} \vdash t : \diamond \qquad \mathcal{R}' \vdash ts : \diamond}{\mathcal{R} * \mathcal{R}' \vdash t \mid ts : \diamond} \text{ (Cons Pool)}$$

Now, the rule for states ensures that there exists a resource $\mathcal{R}$ to satisfy the thread pool ts:

$$\frac{h = \mathcal{R}_{\mathsf{hp}} \qquad \mathcal{R} \vdash ts : \diamond}{\langle h,\ ts\rangle : \diamond}\ \text{(State)}$$

In addition to the rule for states, we now have a rule for threads. Roughly, this rule corresponds to Section 3.5's (State)'s rule but it is modified to model that threads have a fraction of `postJoin<1>` as postconditions. To model this, we introduce *symbolic binary fractions* that represent numbers of the forms 1 or $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$:

$$bit \in \{0,1\} \qquad bits \in \mathsf{Bits} ::= 1 \mid bit, bits \qquad fr \in \mathsf{BinFrac} ::= \mathsf{all} \mid \mathsf{fr}() \mid \mathsf{fr}(bits)$$

Intuitively, we use symbolic binary fractions to speak about finite formulas of the form $r.P\texttt{<1>} * r.P\texttt{<}\frac{1}{2}\texttt{>} * r.P\texttt{<}\frac{1}{8}\texttt{>} * \ldots$. Formally, we define the scalar multiplication $fr \cdot r.P\texttt{<}\pi\texttt{>}$ as follows:

$$\begin{aligned}
\mathsf{all} \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<}\pi\texttt{>}\\
\mathsf{fr}() \cdot r.P\texttt{<}\pi\texttt{>} &= \texttt{true}\\
\mathsf{fr}(1) \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<split}(\pi)\texttt{>}\\
\mathsf{fr}(0, bits) \cdot r.P\texttt{<}\pi\texttt{>} &= \mathsf{fr}(bits) \cdot r.P\texttt{<split}(\pi)\texttt{>}\\
\mathsf{fr}(1, bits) \cdot r.P\texttt{<}\pi\texttt{>} &= r.P\texttt{<split}(\pi)\texttt{>} * \mathsf{fr}(bits) \cdot r.P\texttt{<split}(\pi)\texttt{>}
\end{aligned}$$

For instance, $\mathsf{fr}(1,0,1) \cdot r.P\texttt{<1>} \mathbin{*\!\!-\!\!*} (r.P\texttt{<}\frac{1}{2}\texttt{>} * r.P\texttt{<}\frac{1}{8}\texttt{>})$. The map $[\![\cdot]\!] : \mathsf{BinFrac} \to \mathbb{Q}$ interprets symbolic binary fractions as concrete rationals:

$$[\![\mathsf{all}]\!] \triangleq 1 \quad [\![\mathsf{fr}()]\!] \triangleq 0 \quad [\![\mathsf{fr}(1)]\!] \triangleq \frac{1}{2} \quad [\![\mathsf{fr}(0, bits)]\!] \triangleq \frac{1}{2}[\![\mathsf{fr}(bits)]\!] \quad [\![\mathsf{fr}(1, bits)]\!] \triangleq \frac{1}{2} + \frac{1}{2}[\![\mathsf{fr}(bits)]\!]$$

Now, the rule for threads is as follows:

$$\frac{\begin{array}{c}\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!] \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \quad \mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset\\ \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \quad \Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{\ fr \cdot o\texttt{.postJoin<1>}\ \}\end{array}}{\mathcal{R} \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c) : \diamond}\ \text{(Thread)}$$

In rule (Thread), fr should be bigger than the thread considered's entry in the global join table (condition $\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!]$). This forces joining threads to take back a part of a terminated thread's postcondion which is smaller than the terminated thread's "remaining" postcondition. This comes from the semantics of the `Join` predicate and the semantics of join tables: $\Gamma \vdash (h, \mathcal{P}, \mathcal{J}); s \models \texttt{Join(}e, \pi\texttt{)}$ holds iff $[\![e]\!]^h_s = o$ and $[\![\pi]\!] \leq \mathcal{P}(o, \texttt{join})$. Moreover, by $\mathcal{P} \leq \mathcal{J}$ (see axiom (b) on page 53), we have that $\mathcal{P}(o, \texttt{join}) \leq \mathcal{J}(o)$.

Like in Section 3.5, we have shown the preservation theorem 3 (Section 6.1.2's proof is completed in Section 6.2.2) and we have shown that verified programs satisfy the following properties: null error freeness and partial correctness.

In addition, verified programs are *data race* free. A pair (hc, hc') of head commands is called a *data race* iff $hc = (o.f = v)$ and either $hc' = (o.f = v')$ or $hc' = (\ell = o.f)$ for some o, f, v, v', ℓ.

Theorem 6 (Verified Programs are Data Race Free)**.** *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \to^*_{ct} \langle h,\ ts \mid o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) \mid o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2)\rangle$*, then* (hc_1, hc_2) *is not a data race.*

Proof. Sec Section 6.2.3 □

4.6 Examples of Reasoning with fork/join

4.6.1 Fibonacci

Our first example is a recursive computation of the n-th Fibonacci number that runs recursive calls in new threads. The example is taken from Lea's patterns collection [76, §4.4.1.4]. Although the example is unrealistic, because there are faster non-recursive algorithms to compute Fibonacci numbers, it nicely illustrates how our system works with `fork` and `join`.

The `Fib` sequence shown below is a *divide and conquer* algorithm [76, §4.4.1]. A divide and conquer algorithm has the property that; given a problem of size $n > 0$, it can be divided into multiple problems of size $< n$, and so on recursively (until the size of problems reaches 0). Many computationally intensive problems (such as merge sort or matrix multiplication) can be expressed with divide and conquer algorithms. Generally speaking, we believe our approach fits well to verify such programs.

We do not verify the functional behavior of this algorithm because we would need extra machinery (such as axiomatizing the Fibonacci function which requires user-definable mathematical predicates) but we prove race freedom. The meaning of `preFork` and `postJoin` is as follows:

- `preFork` indicates that field `number` can be written
- `postJoin<p>` indicates that field `number` can be written (if `p` is `1`) or only read (if `p < 1`).

Here is the implementation and specification of class `Fib`:

```
class Fib extends Thread{
 int number;
 pred  preFork = Perm(number,1);
 group postJoin<perm p> = Perm(number,p);
 axiom preFork -* postJoin<1>; // needed to verify run
 requires init;
 ensures  Join(this,1) * preFork@Fib;
 void init(int n){ number = n; }
 requires preFork;
 ensures  postJoin<1>;
 void run(){
  if(!(number < 2)){
   Fib f1 = new Fib; f1.init(number-1);
   Fib f2 = new Fib; f2.init(number-2);
   f1.fork(); f2.fork();
   f1.join(); f2.join();
   number = f1.number + f2.number;
  }
 }
}
```

We now comment method `init` and method `run`.

Method `init` serves as a constructor as predicate `init` in the precondition shows. After method `init`'s return, the `Join` predicate and `preFork@Fib` are available. Even if predicate `preFork` is qualified by `@Fib`, this does not break subtyping. Because method `init` serves as a constructor, predicate `preFork@Fib` (in combination with a `isclassof` predicate, which is emitted in `new`'s postcondition) can be transformed into predicate `preFork` by the axiom (Known Type) (see page 34). This behavior occurs twice in the proof outline for method `run` below.

Method `run` is interesting, because permissions are transferred between threads during its execution. Initially, there is only one thread (the *main* thread), executing its `run` method. At first, the main thread has accessed solely to its `number` field (as specified by `run`'s precondition i.e., `preFork`). Then, two child threads `f1` and `f2` are created but not yet started. After initializing `f1` and `f2`, the main threads has access to `f1` and `f2`'s states. Then, `f1` and `f2` are started: access to their `number` field is transferred from the main thread to `f1` and `f2` themselves. Finally, the main thread waits for `f1` and `f2` to terminate. By joining them, the main thread re-obtain permissions to access `f1.number` and `f2.number`. That is why reading `f1.number` and `f2.number` in the last assignment is correct.

Here is the proof outline for `run`:

```
 { preFork }
 ((Dynamic Type) axiom and modus ponens)
 { preFork@Fib * (preFork@Fib -* preFork) }
 ((Open/Close) axiom and abbreviation of preFork@Fib -* preFork by F)
 { Perm(this.number,1) * F }
 (Because we have Perm(this.number,1), we can read this.number)
if(!(this.number < 2)){
  { Perm(this.number,1) * F }
 Fib f1 = new Fib;
  { Perm(this.number,1) * F * f1.init * Fib classof f1 }
 f1.init(number-1);
  { Perm(this.number,1) * F * f1.preFork@Fib * Fib classof f1 *
    Join(f1,1) }
  ((Known Type) axiom)
  { Perm(this.number,1) * F * f1.preFork * Join(f1,1) }
 Fib f2 = new Fib;
 f2.init(number-2);
  { Perm(this.number,1) * F * f1.preFork * Join(f1,1) *
    f2.preFork * Perm(f2.join,1) }
 f1.fork();
  { Perm(this.number,1) * F * Join(f1,1) * f2.preFork *
    Perm(f2.join,1) }
 f2.fork();
  { Perm(this.number,1) * F * Join(f1,1) * Join(f2,1) }
  (Because we have Join(f1,1), we have full access to the postcondition of f1.join)
 f1.join();
  { Perm(this.number,1) * F * f1.postJoin<1> * Join(f2,1) }
 f2.join();
```

```
  { Perm(this.number,1) * F * f1.postJoin<1> * f2.postJoin<1> }
```
((Open/Close) axiom from left to right on `postJoin` to obtain permissions to read `f1.number` and `f2.number` (details omitted))
```
this.number = f1.number + f2.number;
  { Perm(this.number,1) * F * f1.postJoin<1> * f2.postJoin<1> }
```
(Weakening and (Open/Close) axiom from right to left on `Perm(this.number,1)`)
```
  { preFork@Fib * F }
```
(Modus ponens)
```
  { preFork }
```
(Class `Fib`'s axiom)
```
  { postJoin<1> }
```

Discussion. This example shows our rules for `fork` and `join` in action in a simple setting. When `f1.fork()` is called, the state necessary for thread `f1` to execute is transferred from the parent thread to `f1`: `f1.preFork` is consumed. Dually, when `f1.join()` is called, the parent thread waits for `f1` to terminate and then gets back `f1`'s state (`f1`'s postcondition). Because the parent thread has permission `Join(f1,1)` i.e., full access to `f1`'s postcondition, the parent thread gets back `f1.postJoin<1>`. A similar behavior occurs for thread `f2`.

When proving method `run`, we use class `Fib`'s axiom to establish `run`'s postcondition. Given the definitions of `preFork` and `postJoin`, this axiom might seem useless at first sight. This axiom, however, does not only relate `preFork` and `postJoin`'s definitions in class `Fib`, but also in all possible subclasses of `Fib`. The fact that class `Fib`'s axiom is needed to verify `run` means that `preFork` and `postJoin` have to be related in subclasses of `Fib` for `run` to be safe in class `Fib` *and* in subclasses of `Fib`.

In method `run` of class `Fib`, threads do not share the postcondition of a terminated thread. In other words, whenever rule (Mth) is applied with $m =$ `join`, π is instantiated by 1. Thus, in this case, it is not necessary for the implementation of `postJoin` to be a group (i.e., to satisfy `postJoin<p> *-* postJoin<p/2> * postJoin<p/2>` for all `p`).

We could support both arbitrary implementations of `postJoin` and multiple joiners if we introduced a `run`-method modifier "`multi-join`" such that only `multi-join` `run`-methods may have multiple resource-splitting joiners and must have implementations of `postJoin` that are groups. It would also be important to require that methods that override `multi-join` methods are again `multi-join`. This would allow (1) thread classes that are meant to be joined by a single thread (in these classes, `postJoin` could be an arbitrary predicate, not a group) and (2) thread classes that can be joined by one or more threads (in these classes, `postJoin` would have to be a group).

4.6.2 Replication of Databases[2]

Our second example sketches how to specify and verify a system that updates a database and then replicates (in parallel) the database to multiple mirrors. Interfaces `Database` and `Mirror` are as follows:

```
interface Database{
 group state<perm p>;
```

[2]This example was suggested to me by Jean-Christophe Bach.

```
 requires state<1>; ensures state<1>;
 void update();
}
interface Mirror{
 pred state<perm p>;
 requires init; ensures state<1>;
 void init();
 requires state<1> * d.state<p>; ensures state<1> * d.state<p>;
 void replicate(Database d);
}
```

In both classes, predicate `state` represents the internal state of objects. To update a database, `state<1>` is required (i.e., write access). To replicate a database to a mirror, `state<1>` is required on the mirror but only `state<p>` (i.e., readonly access) is required on the database.

Below, we present a dedicated thread to update a database. For later convenience, we use the `multi-join` modifier discussed in Section 4.6.1 to indicate that `postJoin`'s definition in class `DatabaseUpdater` is a group i.e., a `DatabaseUpdater` can be soundly joined by multiple threads.

```
multi-join class DatabaseUpdater<Database updtdb> extends Thread{
 Database d;
 pred preFork = PointsTo(d,1,updtdb) * updtdb.state<1>;
 public pred postJoin<perm p> = PointsTo(d,p,updtdb) * updtdb.state<p>;
 axiom preFork -* postJoin<1>; // needed to verify run
 requires init * f.state<1> * f==updtdb;
 ensures  preFork@DatabaseUpdater * Join(this,1);
 void init(Database f){ this.d = f; }
 requires preFork; ensures postJoin<1>;
 void run(){ d.update(); }
 requires postJoin<p>;
 ensures  result.state<p> * (result.state<p> -* postJoin<p>);
 void getDatabase(){ d }
}
```

Class `DatabaseUpdater` is parameterized by the database being updated `updtdb`. Parameter `updtdb` expresses that field `d` is – in some sense – final, i.e., it is assigned only once. This is enforced by the definitions of `preFork` and `postJoin`. For `preFork` and `postJoin` to hold, field `d` *must* contain object `updtdb`. That is why we need `f==updtdb` in `init`'s precondition. We will use this specification's style a lot in class `DatabaseReplicater` below and we discuss this usage later in this section. As class `DatabaseUpdater` extends `Thread` and is `multi-join`, `postJoin` implementation is a group. To show that `postJoin` is a group, we use that class `Database`'s predicate `state` is also a group.

Like class `Fib` before, class `DatabaseUpdater` uses an axiom to constrain extensions of `preFork` and `postJoin` in possible subclasses. As indicated in Java comments, this

axiom (or constraint) is required for method `run` to be correct. We highlight that this axiom does not simply impose a constraint on `preFork` and `postJoin`'s definitions in class `DatabaseUpdater` but also imposes a constraint on `preFork` and `postJoin`'s definitions in `DatabaseUpdater`'s *subclasses*.

The postcondition of method `getDatabase` shows a use of the magic wand that we often encounter (see also method `next` in Section 3.6.2's `Iterator`). Generally, a postcondition of the form F `*` `(`F `-*` G`)` where F gives access to the receiver's internal representation and G gives access to the receiver means that clients are – temporarily or indefinitely – allowed to access an object's internal representation. In `getDatabase`'s case, this means that (1) access to the internal database is given to clients: `result.state<p>` is ensured and (2) clients may give up access to the internal database to recover access to the `DatabaseUpdater` thread (`result.state<p> -* postJoin<p>`).

Generally speaking, postconditions of the form described above give clients two choices: either they temporarily access the receiver's internal representation and get access to the receiver back by using modus ponens, either they indefinitely access the receiver's internal representation. Alternatively, we can specify policies where access to the receiver's internal representation is persistent: access to the receiver can never be recovered. Such a policy can be described with an axiom of the form F `-*` G (where F and G play similar roles as above). Interface `Iterator`'s axiom in Section 3.6.2 exemplifies such a policy.

We provide a dedicated thread to replicate a database to one mirror:

```
class DatabaseReplicater<perm q,
                         DatabaseUpdater src,
                         Mirror dest> extends Thread{
 DatabaseUpdater du;
 Mirror m;
 public pred state<perm p> = PointsTo(du,p,src) * PointsTo(m,p,dest) *
                             dest.state<p>;
 pred preFork = state<1> * Join(src,q);
 public pred postJoin<perm p> = state<q> * src.postJoin<q>;
 axiom (state<1> * Join(src,q)) -* preFork;          // to verify init
 axiom (state<q> * src.postJoin<q>) -* postJoin<1>; // to verify run
 requires init * Join(f,q) * n.state<1> * f==src * n==dest;
 ensures preFork * Join(this,1);
 void init(DatabaseUpdater f, Mirror n){ this.du = f; this.m = n; }
 requires preFork; ensures postJoin<1>;
 void run(){ du.join();
             Database data = du.getDatabase();
             m.replicate(data); }
}
```

Class `DatabaseReplicater` is not `multi-join`. On one hand, this is convenient because the implementation of `postJoin` is not a group. On the other hand, for this to be sound, we have to make sure that wherever rule (Mth) is applied (with m = `join`) on a thread whose static type is `DatabaseReplicater`, π is instantiated by 1. The implementation of `postJoin`'s in `DatabaseReplicater` is peculiar because it ignores the

permission parameter `p`. That is why class `DatabaseReplicater`'s second axiom is sound: `postJoin`'s parameter is meaningless.

Method `run` of class `DatabaseReplicater` is the most interesting method. First, it waits for the database updater to terminate (`du.join()`). Then, the database is retrieved (`data = du.getDatabase()`). Finally, the database is replicated to the mirror (`m.replicate(data)`). On the level of access permissions, a similar behavior occurs : the database is unpacked from the `DatabaseUpdater` thread, then the database is replicated to the mirror, and it is packed again in the `DatabaseUpdater` thread. This means that the `DatabaseReplicater` thread has temporary access to the database during `run`, but before `run` terminates, access to the database is given back to the `DatabaseUpdater` thread. This is reflected by the proof outline below, that shows correctness of method `run` of class `DatabaseReplicater`:

```
 { preFork }
 ((Dynamic Type) axiom and modus ponens)
 { preFork@DatabaseReplicater * (preFork@DatabaseReplicater -* preFork) }
 ((Open/Close) axiom and weakening)
 { state<1> * Join(src,q) }
 ((Dynamic Type) axiom and modus ponens)
 { state@DatabaseReplicater<1> * (state@DatabaseReplicater<1> -* state<1>) *
   Join(src,q) }
 ((Open/Close) axiom and abbreviating state@DatabaseReplicater<1> -* state<1> by F)
 { PointsTo(du,1,src) * PointsTo(m,1,dest) * dest.state<1> * Join(src,q) * F }
du.join()
 { PointsTo(du,1,src) * PointsTo(m,1,dest) * dest.state<1> * du.postJoin<q> * F }
data = du.getDatabase();
 { PointsTo(du,1,src) * PointsTo(m,1,dest) * dest.state<1> * data.state<q> *
   (data.state<q> -* src.postJoin<q>) * F }
 (Abbreviating data.state<q> -* src.postJoin<q> by G)
 { PointsTo(du,1,src) * PointsTo(m,1,dest) * dest.state<1> * data.state<q> *
   F * G }
m.replicate(data)
 { PointsTo(du,1,src) * PointsTo(m,1,dest) * dest.state<1> * data.state<q> *
   F * G }
 ((Open/Close) axiom)
 { state@DatabaseReplicater<1> * data.state<q> * F * G }
 (Modus ponens)
 { state<1> * data.state<q> * G }
 (Modus ponens)
 { state<1> * src.postJoin<q> }
 (Class DatabaseReplicater's second axiom)
 { postJoin<1> }
```

Above, the second application of modus ponens packs the database back into the `DatabaseUpdater` thread.

The following piece of code shows how to to update and replicate a database to two mirrors:

```
requires d.state<1> * m1.state<1> * m2.state<1>;
ensures  d.state<1> * m1.state<1> * m2.state<1>;
void replicateTwice(Database d, Mirror m1, Mirror m2){
 Thread t = new DatabaseUpdater<d>;
```

```
 t.init(d); t.start();
 Thread u1,u2;
 u1 = new DatabaseReplicater<1/2,t,m1>;
 u2 = new DatabaseReplicater<1/2,t,m2>;
 u1.init(t,m1); u2.init(t,m2);
 u1.start(); u2.start();
 u1.join(); u2.join();
}
```

The following proof outline (where, for space reasons, we omit the `classof` predicates) demonstrates correctness of method `replicateTwice`:

```
 { d.state<1> * m1.state<1> * m2.state<1> }
Thread t = new DatabaseUpdater<d>;
 { d.state<1> * m1.state<1> * m2.state<1> * t.init }
t.init(d);
 { m1.state<1> * m2.state<1> * t.preFork@DatabaseUpdater * Join(t,1) }
 ((Known Type) axiom)
 { m1.state<1> * m2.state<1> * t.preFork * Join(t,1) }
t.start();
 { m1.state<1> * m2.state<1> * Join(t,1) }
Thread u1,u2; u1 = new DatabaseReplicater<1/2,t,m1>;
 { m1.state<1> * m2.state<1> * Join(t,1) * u1.init }
 ((Split/Merge Join) axiom)
 { m1.state<1> * m2.state<1> * Join(t,1/2) * Join(t,1/2) * u1.init }
u1.init(t,m1);
 ((Known Type) axiom)
 { m2.state<1> * Join(t,1/2) * u1.preFork * Join(u1,1) }
u2 = new DatabaseReplicater<1/2,t,m2>; u2.init(t,m2);
 ((Known Type) axiom)
 { u1.preFork * Join(u1,1) * u2.preFork * Join(u2,1) }
u1.start(); u2.start();
 { Join(u1,1) * Join(u2,1) }
u1.join(); u2.join();
 { u1.postJoin<1> * u2.postJoin<1> }
 ((Known Type) axiom)
 { u1.postJoin@DatabaseReplicater<1> * u2.postJoin@DatabaseReplicater<1> }
 (Using that postJoin is public in class DatabaseReplicater)
 { u1.state<1> * t.postJoin<1/2> * u2.state<1> * t.postJoin<1/2> }
 (Class Thread's axiom on postJoin)
 { u1.state<1> * t.postJoin<1> * u2.state<1> }
 ((Known Type) axiom)
 { u1.state@DatabaseReplicater<1> * t.postJoin<1> *
   u2.state@DatabaseReplicater<1> }
 (Using that state is public in class DatabaseReplicater and weakening)
 { m1.state<1> * t.postJoin<1> * m2.state<1> }
 ((Known Type) axiom)
 { m1.state<1> * t.postJoin@DatabaseUpdater<1> * m2.state<1> }
 (Using that postJoin is public in class DatabaseUpdater and weakening)
 { m1.state<1> * d.state<1> * m2.state<1> }
```

Discussion. First, we highlight that the two `DatabaseReplicater` threads join the same `DatabaseUpdater` thread. If we did not allow multiple joiners, we would not be able to verify this example. Second, we make heavy use of class parameters, axioms, and `public` modifiers. By doing so, we partially break the abstraction provided by object-oriented programming, because parts of the implementations of predicates are revealed. This is, however, dictated by the example we use:

In method `replicateTwice`'s precondition, `d`, `m1` and `m2` are visible to the client. Inside method `replicateTwice`, however, the database `d` is "packed" into the `DatabaseUpdater` thread and mirrors `m1` and `m2` are packed into `DatabaseReplicater` threads. Because method `replicateTwice`'s postcondition mentions `d`, `m1`, and `m2`, we have to "unpack" these objects to prove `replicateTwice`'s postcondition. That is why we have to keep track of how objects are stored into each others: that is the role of class parameters. An alternative would be to use final fields and to keep track of informations such as `m1.du==d & u1.d==du & u2.d==du` in method contracts. Such an assertion is correct as long as fields on the right of a "`.`" are final. We believe this is a nice way to simplify contracts and it is realistic: in real world Java programs, data is often passed to threads by storing it into final fields. We have shown how to handle final fields in our previous work [54]. While this makes specifications easier to write, the overall verification system gets more complicated.

Similarly, `axiom`s tell clients that they can throw away an object to obtain access to the object's internal representation. For example, the `public` modifier in `postJoin`'s definition in class `DatabaseUpdater` plays this role. Informally, it means that "one can throw away a terminated `DatabaseUpdater` thread to get access to its internal database".

While we admit that our specifications are quite verbose for such an example, we highlight that it would be hard or impossible to express this example in formalisms that do not allow *ownership transfer*. In our setting, public predicates and class axioms give different (recall the distinction between `public` and `axiom` explained at the end of Section 3.6.1) ways to express possible ownership transfers.

4.7 Related Work and Conclusion

Related Work. Recent approaches in classical Hoare logics [69, 12, 77] studied verification of programs with fork and join as concurrency primitives. In Jacobs et al.'s work [69], objects can be in two states: unshared or shared. Unshared objects can only be accessed by the thread that created them; while shared objects can be accessed by all threads, provided these threads synchronize on this object. While this policy is simple, it is too restrictive: an object cannot be passed by one thread to another thread without requiring the latter thread to synchronize on this object. On the upside, Jacobs et al.'s verification system uses automatic standard SMT solvers, while – for automatic verification – we would require dedicated separation logic based provers. Like Jacobs, Leino and Müller [77] showed a verification system that uses SMT solvers. They, however, do not impose a programming model: they use fractional permissions to handle concurrency. They do not support multiple readonly joiner threads.

Separation logic based approaches for parallel programs [84, 15] focused on a theoretically elegant, but unrealistic, parallel operator. Notable exceptions are Hobor et al. [58] and Gotsman et al. [51] who studied (concurrently to us) Posix threads for C-like programs. Contrary to us, Hobor et al. do not model join as a native method, instead they

require programmers to model join with locks. For verification purposes, this means that Hobor et al. would need extra facilities to make reasoning about fork/join as simple as we do. Gotsman et al.'s work is very similar to Hobor et al.'s work.

Conclusion. We showed a Java-like language with fork and join as concurrency primitives. The key contributions of this chapter are as follows: (1) we presented verification rules for fork and join in an object-oriented language with inheritance; (2) we showed how verification rules for fork and join can be expressed with general purpose abstract predicates [88], groups, and class axioms; and (3) we provided verification rules that allow both single write-joiner and multiple readonly-joiners (which Gotsman et al. [51] do not support).

In the next chapter, we extend our model language and our verification system to deal with reentrant locks i.e., Java's basic primitive for synchronization mechanism.

Chapter 5

Separation Logic for Reentrant Locks

In this chapter, we present verification rules for Java's reentrant locks. Together with `fork` and `join`, reentrant locks are a crucial feature of Java for multithreaded programs. Generally speaking, locks serve to synchronize threads and to control exclusive access to resources that cannot be shared.

Reentrant locks can be acquired more than once by the same thread. This differs from Posix threads that block if they acquire a lock twice. On one hand, reentrant locks are a convenient tool for programmers because code does not need to check if a lock is already acquired before trying to acquire it. On the other hand, reentrant locks require extra machinery in the verification system, because initial acquirements have to be distinguished from reentrant acquirements.

This chapter is structured as follows: in Section 5.1 we present how single-entrant locks are handled in separation logic, in Section 5.2 we show how to modify Section 4.2's Java-like language to model reentrant locks, in Section 5.3 we describe new separation logic formulas used to handle reentrant locks, in Section 5.4 we present verification rules for reentrant locks, and in Section 5.5 we describe what are verified programs. We present three examples of verified programs in Section 5.6: an implementation of mathematical sets that exhibits typical usage of lock reentrancy in the Java library, an implementation of the worker thread pattern [76, 4.1.4], and a challenging lock-coupling algorithm. Finally, we discuss related work and conclude in Section 5.7.

Note: The work from this chapter has been done in collaboration with Christian Haack and Marieke Huisman. It is published in the Asian Symposium on Programming Languages and Systems (APLAS 2008) [53].

5.1 Separation Logic and Locks

Separation logic for programming language with locks as a concurrency primitive has been first explored by O'Hearn [84]. O'Hearn elegantly adapted an old idea from concurrent programs with shared variables [3]. Each lock is associated with a *resource invariant* which describes the part of the heap that the lock guards. When a lock is acquired, it

lends its resource invariant to the acquiring thread. Dually, when a lock is released, it takes back its resource invariant from the releasing thread. This is formally expressed by the following Hoare rules:

$$\frac{I \text{ is } x\text{'s resource invariant}}{\{\texttt{true}\}x.\texttt{lock()}\{I\}} \qquad \frac{I \text{ is } x\text{'s resource invariant}}{\{I\}x.\texttt{unlock()}\{\texttt{true}\}}$$

While these rules are sound for single-entrant locks, they are unsound for reentrant locks, because they allow to "duplicate" a lock's resource invariant:

```
 { true }
x.lock(); // I is x's resource invariant
 { I }
x.lock();
 { I * I } ← wrong!
```

To recover soundness in the presence of reentrant locks, we design proof rules that distinguish between initial acquirement and reentrant acquirement of locks. This allows to transfer a lock's resource invariant to an acquiring thread only at initial acquirement. Compared to the literature [84, 58, 51], where simple while languages and C-like languages are studied, we also handle inheritance.

5.2 A Java-like Language with Reentrant Locks

We show how to modify the syntax and the semantics of Section 4.2's language to model reentrant locks.

Syntax. We extend the list of head commands defined in Section 2.1 as follows:

$$hc \in \mathsf{HeadCmd} \quad ::= \quad \ldots \mid v.\texttt{lock()} \mid v.\texttt{unlock()} \mid \ldots$$

The meaning of `lock` and `unlock` is exactly like `synchronized` blocks in Java. We choose `lock` and `unlock` as locking primitives because Java 5 provides lock *objects* that provide a behavior similar to our primitives. In addition, `lock` and `unlock` primitives are more expressive than `synchronized` blocks. First, `synchronized` blocks forbid to lock and unlock an object in different methods. Second, `synchronized` blocks enforce proper nesting of `lock` and `unlock` commands. To exemplify the second statement, consider the following code:

```
o.lock(); q.lock(); o.unlock(); q.unlock();
```

Such a behavior cannot be reproduced with `synchronized` blocks.

Like class invariants must be initialized before method calls, resource invariants must be initialized before the associated locks can be acquired. In O'Hearn's simple concurrent language [84], the set of locks is static and initialization of resource invariants is achieved in a global initialization phase. This is not possible when locks are created dynamically. Conceivably, we could tie the initialization of resource invariants to the end of object constructors. However, this is problematic because Java's object constructors are free to leak references to partially constructed objects (e.g., by passing `this` to other methods). Thus, in practice we have to distinguish between initialized and uninitialized objects semantically. Furthermore, a semantic distinction enables late initialization of resource

invariants, which can be useful for objects that remain thread-local for some time before getting shared among threads.

We distinguish between *fresh* locks and *initialized* locks. A fresh lock does not yet guard its resource invariant: a fresh lock is not ready to be acquired yet. An initialized lock, however, is ready to be acquired. Initially, locks are fresh and they might become initialized later. We require programmers to explicitly change the state of locks (from fresh to initialized) with a `commit` command:

$$sc \in \mathsf{SpecCmd} \quad ::= \quad \ldots \mid \pi.\texttt{commit} \mid \ldots$$

Operationally, π.`commit` is a no-op; semantically it checks that π is fresh and changes π's state to initialized. For expressiveness purposes, `commit`'s receiver ranges over specification variables, which include both program variables and logical variables (such as class parameters). In real-world Java programs, a possible default would be to add a `commit` command at the end of constructors. Another possibility would be to infer `commit` commands automatically.

Like in Java, we assume that class tables always contain the following class declaration:

```
class Object {
 pred inv = true;
 final void wait();
 final void notify();
}
```

The distinguished `inv` predicate assigns to each lock a resource invariant. The definition `true` is a default and objects meant to be used as locks should extend `inv`'s definition in subclasses of `Object`. As usual [84], the resource invariant o.`inv` can be assumed when o's lock is acquired non-reentrantly and must be established when o's lock is released with its reentrancy level dropping to 0. Regarding the interaction with subclassing, there is nothing special about `inv`. It is treated just like other abstract predicates.

The methods `wait` and `notify` do not have Java implementations, but are implemented natively. To model this, our operational semantics treats them in a special way (see $\rightarrow$'s definition on page 71). Intuitively, these methods behave as follows:

If o.`wait()` is called when object o is locked at reentrancy level n, then o's lock is released and the executing thread temporarily stops executing. When a thread calls o.`notify()`, one thread that is stopped (because this thread called o.`wait()` before) resumes and starts competing for o's lock. When a resumed thread reacquires o's lock, its previous reentrancy level is restored.

We do not provide contracts for `wait` and `notify` yet, but will do so in Section 5.4. On one hand, we do not put `wait` and `notify` in our set of commands, because we can specify them with the contracts available to programmers. On the other hand, we put `lock`, `unlock`, and `commit` in our set of commands. This is because the Hoare rules for these methods cannot be expressed with the syntax of contracts available to programmers: we need extra expressivity (see Section 5.4).

In addition to `wait` and `notify`, Java provides method `notifyAll` to notify all threads waiting on an object. We do not include `notifyAll` in our verification system because it can be treated exactly like `notify`.

Runtime Structures. To represent locks in the operational semantics, we use a *lock table*. *Lock tables* map objects o to either the symbol free, or to the thread object that currently holds o's lock and a number that counts how often it currently holds this lock:

$$l \in \mathsf{LockTable} = \mathsf{ObjId} \rightharpoonup \{\mathsf{free}\} \uplus (\mathsf{ObjId} \times \mathbb{N})$$

Compared to Section 4.2, states are extended to include a heap, a lock table, and a thread pool:

$$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{LockTable} \times \mathsf{ThreadPool}$$

Initialization. We modify Section 4.2's definition of the initial state of a program. Initially, the lock table of a program is empty (hence the second $\emptyset$):

$$\mathsf{init}(c) = \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\},\ \emptyset,\ \mathsf{main\ is}\ (\emptyset\ \mathsf{in}\ c)\rangle$$

We modify the *operational semantics* defined in Section 4.2 to deal with locks. Except that a lock table is added, most of the existing cases of the operational semantics are left untouched.

To represent states in which threads are waiting to be notified, we could associate each object with a set of waiting threads (the "wait set"). However, we prefer to avoid introducing yet another runtime structure, and therefore represent waiting states syntactically:

$$hc \ ::= \ \ldots \ \mid \ o.\texttt{waiting}(n) \ \mid \ o.\texttt{resume}(n) \ \mid \ \ldots$$

Restriction: These clauses must not occur in source programs.

Here are the intuitive semantics of these head commands:

- $o.\texttt{waiting}(n)$: If thread p's head command is $o.\texttt{waiting}(n)$, then p is waiting to be notified to resume competition for o's lock at reentrancy level n.
- $o.\texttt{resume}(n)$: If thread p's head command is $o.\texttt{resume}(n)$, then p has been notified to resume competition for o's lock at reentrancy level n, and is now competing for this lock.

Below, we list existing cases that are slightly modified ((Red New) and (Red Call)) and cases that are added ((Red Lock), (Red Unlock), (Red Wait), (Red Notify), (Red Skip Notify), and (Red Resume)):

State Reductions, $st \rightarrow_{ct} st'$**:**

$\ldots$

(Red New) $\quad o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o] \quad l' = l[o \mapsto \mathsf{free}]$

$\langle h,\ l\ ,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c)\rangle \ \rightarrow\ \langle h',\ l'\ ,\ ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red Call) $\quad m \notin \{\texttt{fork}, \texttt{join}, \texttt{wait}, \texttt{notify}\}$

$h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\iota_0, \bar{\iota}).c_m \quad c' = c_m[o/\iota_0, \bar{v}/\bar{\iota}]$

$\langle h,\ l\ ,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}o.m(\bar{v});\ c)\rangle \ \rightarrow\ \langle h,\ l\ ,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \leftarrow c';\ c)\rangle$

(Red Lock) $\quad (l(o) = \mathsf{free},\ l' = l[o \mapsto (1,p)])$ or $(l(o) = (n,p),\ l' = l[o \mapsto (n+1,p)])$

$\langle h,\ l,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ o\texttt{.lock();}\ c)\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c)\rangle$

(Red Unlock) $\quad l(o) = (n,p) \quad n = 1 \Rightarrow l' = l[o \mapsto \mathsf{free}] \quad n > 1 \Rightarrow l' = l[o \mapsto (n-1,p)]$

$\langle h,\ l,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ o\texttt{.unlock();}\ c)\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c)\rangle$

(Red Wait) $l(o) = (n, p) \quad l' = l[o \mapsto \mathsf{free}]$
$\langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } \ell = o.\texttt{wait}();\ c)\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid p \text{ is } (s \text{ in } o.\texttt{waiting}(n);\ o.\texttt{resume}(n);\ c)\rangle$

(Red Notify) $l(o) = (n, p)$
$\langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } \ell = o.\texttt{notify}();\ c) \mid q \text{ is } (s' \text{ in } o.\texttt{waiting}(n');\ c')\rangle \ \rightarrow$
$\langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } c) \mid q \text{ is } (s' \text{ in } c')\rangle$

(Red Skip Notify) $l(o) = (n, p)$
$\langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } \ell = o.\texttt{notify}();\ c)\rangle \ \rightarrow\ \langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } c)\rangle$

(Red Resume) $l(o) = \mathsf{free} \quad l' = l[o \mapsto (n, p)]$
$\langle h,\ l,\ ts \mid p \text{ is } (s \text{ in } o.\texttt{resume}(n);\ c)\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid p \text{ is } (s \text{ in } c)\rangle$

...

Remarks.

- Rule (Red Lock) distinguishes two cases: (1) lock o is acquired for the first time ($l(o) = \mathsf{free}$) and (2) lock o is acquired reentrantly ($l(o) = (n, p)$).
- Similarly, rule (Red Unlock) distinguishes two cases: (1) lock o's reentrancy level decreases but o remains acquired ($l(o) = (n, p)$ and $n > 1$) and (2) lock o is released ($l(o) = (1, p)$).
- Rule (Red Wait) fires only if the thread considered previously acquired `wait`'s receiver. In this case, `wait`'s receiver is released and the thread goes in the `waiting` state. The thread's reentrancy level is stored in `waiting`'s argument.
- Like rule (Red Wait), rules (Red Notify) and (Red Skip Notify) fire only if the thread considered previously acquired `notify`'s receiver. Rule (Red Notify) fires if there is a thread waiting on `notify`'s receiver. In this case, the waiting thread is resumed. If there is no thread waiting on `notify`'s receiver, rule (Red Skip Notify) fires. In this case, the call to `notify` has no effect on other threads.
- In Java, if o`.wait()` and o`.notify()` are called by a thread that does not hold o, an `IllegalMonitorState` exception is raised. In our semantics, this is modeled by being stuck. In Section 5.4, we will give contracts for `wait` and `notify` that ensure verified programs are never stuck when calling `wait` or `notify`. In another words, verified programs would never throw an `IllegalMonitorState` exception in Java's semantics.
- Rule (Red Resume) resumes a thread that previously waited on some lock. The thread's reentrancy level is restored.

5.3 Separation Logic for Reentrant Locks

In this section, we describe the new formulas that we add to the specification language of Section 4.3.

In separation logic for single-entrant locks [84], locks can be acquired unconditionally. For reentrant locks, on the other hand, it seems unavoidable that the proof rule for acquiring a lock distinguishes between initial acquires and reacquires. This is needed because it is quite obviously unsound to simply assume the resource invariant after a re-acquire. Thus, a proof system for reentrant locks must keep track of the locks that the current thread holds. To this end, we enrich our specification language:

$$\begin{array}{lll} \pi \in \mathsf{SpecVal} & ::= & \ldots \mid \texttt{nil} \mid \pi \cdot \pi \mid \ldots \\ F \in \mathsf{Formula} & ::= & \ldots \mid \texttt{Lockset}(\pi) \mid \pi\ \texttt{contains}\ e \mid \ldots \end{array}$$

Here is the informal semantics of the new specification values and formulas:

- `nil`: the empty multiset.
- $\pi \cdot \pi'$: the multiset union of multisets π and π'.
- `Lockset(`π`)`: π is the multiset of locks held by the current thread. Multiplicities record the current reentrancy level. *(non-copyable)*
- π `contains` e: multiset π contains object e. *(copyable)*

We classify the new formulas (of which there will be two more) into *copyable* and *non-copyable* ones. Copyable formulas represent *persistent state properties* (i.e., properties that hold forever, once established), whereas non-copyable formulas represent *transient state properties* (i.e., properties that hold temporarily). For copyable F, we postulate the axiom `(`G `&` F`) -* (`G `*` F`)`, whereas for non-copyable formulas we postulate no such axiom. Note that this axiom implies F `-* (`F `*` F`)`, hence the term "copyable". As indicated above, π `contains` e is copyable, whereas `Lockset(`π`)` is not.

Initial Locksets. When verifying the body of `Thread.run()`, we assume `Lockset(nil)` as a precondition.

Initializing Resource Invariants. As explained before, resource invariants must be initialized before the associated locks can be acquired. We use the specification command `commit` to indicate where a lock changes from the fresh state to the initialized state. Because we do not tie initialization to a specific program point (such as the end of constructors), we also have to keep track of the state of locks in our verification system. To this end, we introduce two more formulas:

$$F \in \mathsf{Formula} \quad ::= \quad \ldots \mid e\texttt{.fresh} \mid e\texttt{.initialized} \mid \ldots$$

Restriction: e`.initialized` must not occur in negative positions.

- e`.fresh`: e's resource invariant is not yet initialized. *(non-copyable)*
- e`.initialized`: e's resource invariant has been initialized. *(copyable)*

Because e`.initialized` is copyable, `initialized` formulas can "spread" to all threads, allowing all threads to try to acquire locks (in Section 5.4, we will see that `initialized` appears in the precondition of the Hoare rules for lock acquirement).

Types. We add a type to represent locksets:

$$T \quad ::= \quad \ldots \mid \texttt{lockset} \mid \ldots$$

To accommodate this new type, we update Section 3.2.2's typing rule for good types:

Good Types, $\Gamma \vdash T : \diamond$:

$$\ldots \quad \frac{T \in \{\texttt{void}, \texttt{int}, \texttt{bool}, \texttt{perm}, \texttt{lockset}\}}{\Gamma \vdash T : \diamond} \quad \text{(Ty Primitive)} \quad \ldots$$

It is convenient to allow using objects as singleton locksets (rather than introducing explicit syntax for converting from objects to singleton locksets). Hence, we postulate `Object` $<:$ `lockset`.

Because we allow arbitrary specification values (including locksets) as type parameters, we postulate that types with semantically equal type parameters are type-equivalent. Technically, we let $\simeq$ be the least equivalence relation on specification values that satisfies the standard multiset axioms:

Equivalence of Specification Values: $\pi \simeq \pi$

$$\texttt{nil} \cdot \pi \simeq \pi \qquad \pi \cdot \pi' \simeq \pi' \cdot \pi \qquad (\pi \cdot \pi') \cdot \pi'' \simeq \pi \cdot (\pi' \cdot \pi'')$$

Then we postulate that $t\texttt{<}\bar{\pi}\texttt{>} <: t\texttt{<}\bar{\pi}'\texttt{>}$ when $\bar{\pi} \simeq \bar{\pi}'$. Now, we extend Section 4.3's judgment for well-typed formulas:

Well-typed Formulas, $\Gamma \vdash F : \diamond$**:**

$$\ldots \quad \frac{\Gamma \vdash \pi : \texttt{lockset}}{\Gamma \vdash \texttt{Lockset(}\pi\texttt{)} : \diamond}\ (\text{Form Lockset}) \qquad \frac{\Gamma \vdash \pi, e : \texttt{lockset}, \texttt{Object}}{\Gamma \vdash \pi\ \texttt{contains}\ e : \diamond}\ (\text{Form Contains}) \qquad \frac{\Gamma \vdash e : \texttt{Object}}{\Gamma \vdash e\texttt{.fresh} : \diamond}\ (\text{Form Fresh})$$

$$\frac{\Gamma \vdash e : \texttt{Object}}{\Gamma \vdash e\texttt{.initialized} : \diamond}\ (\text{Form Initialized}) \qquad \ldots$$

Resources. To express the semantics of the new formulas, we need to extend resources with three new components. From now on, resources are 6-tuples of a heap, a permission table, a join table, an *abstract lock table* $\mathcal{L} \in \mathsf{ObjId} \rightharpoonup \mathsf{Bag}(\mathsf{ObjId})$, a *fresh set* $\mathcal{F} \subseteq \mathsf{ObjId}$, and an *initialized set* $\mathcal{I} \subseteq \mathsf{ObjId}$.

Abstract lock tables map thread identifiers to locksets. Like permission tables that are abstraction of heaps, abstract lock tables are abstraction of lock tables. The compatibility relation captures that distinct threads cannot hold the same lock (we use $\sqcap$ to denote bag intersection, $\sqcup$ for bag union, and $[\,]$ for the empty bag):

$$\mathcal{L}\#\mathcal{L}' \text{ iff } \begin{cases} \mathsf{dom}(\mathcal{L}) \cap \mathsf{dom}(\mathcal{L}') = \emptyset \\ (\forall o \in \mathsf{dom}(\mathcal{L}), p \in \mathsf{dom}(\mathcal{L}'))(\mathcal{L}(o) \sqcap \mathcal{L}'(p) = [\,]) \end{cases} \qquad \mathcal{L} * \mathcal{L}' \triangleq \mathcal{L} \cup \mathcal{L}'$$

Fresh sets $\mathcal{F}$ keep track of allocated but not yet initialized objects, while *initialized sets* $\mathcal{I}$ keep track of initialized objects. We define $\#$ for fresh sets as disjointness to mirror that o.`fresh` is non-copyable, and for initialized sets as equality to mirror that o.`initialized` is copyable:

$$\begin{array}{llll} \mathcal{F}\#\mathcal{F}' & \text{iff} & \mathcal{F} \cap \mathcal{F}' = \emptyset & \mathcal{F} * \mathcal{F}' \triangleq \mathcal{F} \cup \mathcal{F}' \\ \mathcal{I}\#\mathcal{I}' & \text{iff} & \mathcal{I} = \mathcal{I}' & \mathcal{I} * \mathcal{I}' \triangleq \mathcal{I}\ (= \mathcal{I}') \end{array}$$

We require resources to satisfy the following axioms (in addition to Section 4.3's axioms):

(a) $\mathcal{F} \cap \mathcal{I} = \emptyset$.
(b) If $o \in \mathcal{L}(p)$ then $o \in \mathcal{I}$.

Axiom (a) ensures that our interpretation of fresh sets and initialized sets makes sense: an object can never be both fresh *and* initialized. Axiom (b) ensures that locked objects are initialized.

As usual, we define projection operators:

$$(h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\mathsf{lock}} \triangleq \mathcal{L} \qquad (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\mathsf{fresh}} \triangleq \mathcal{F} \qquad (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I})_{\mathsf{init}} \triangleq \mathcal{I}$$

Predicate Environments. In addition to Section 4.3's axioms, we require predicate environments to satisfy the following axiom:

(a) If $(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}), r, \bar{\pi}') \in \mathsf{Dom}(\kappa)$,
then $\mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}), r, \bar{\pi}') \leq \mathcal{E}(\kappa)(\bar{\pi}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I} \cup \{o\}), r, \bar{\pi}')$.

Axiom (a) is a technical condition used to update the global initialized set when an object (here o) is committed.

Semantics of Values. Before defining the semantics of formulas, we need to extend the semantics of values to locksets. Recall that SemVal is the set of semantics values (defined in Section 3.2.5). Formally, $\mathsf{SemVal} = \{\texttt{null}\} \cup \mathsf{ObjId} \cup \mathsf{Int} \cup \mathsf{Bool} \cup (0, 1]$. We extend this set to include semantic domains for locksets. The resulting set of semantic values is defined as follows:

$$\mu \in \mathsf{SemVal} \quad \triangleq \quad (\{\texttt{null}\} \cup \mathsf{ObjId} \cup \mathsf{Int} \cup \mathsf{Bool} \cup (0, 1] \cup \mathsf{Bag}(\mathsf{ObjId}))/\equiv$$

where $\equiv$ is the least equivalence relation on SemVal such that $o \equiv [o]$ for all object ids o. That is, $\equiv$ is the least equivalence relation that identifies object identifiers with singleton bags.

The following typing rule extends typing to values representing locksets:

$$\frac{\mu \in \mathsf{Bag}(\mathsf{ObjId})}{\Gamma \vdash \mu : \texttt{lockset}}$$

Let $\mathsf{WellTypClSpecVal}$ be the set of well-typed, closed specification values:

$$\mathsf{WellTypClSpecVal} \triangleq \{\ \pi \mid (\exists \Gamma, T)(\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \text{ and } \Gamma \vdash \pi : T)\ \}$$

To define the semantics of well-typed, closed specification values, we simply define the semantics of the two new specification values:

$$[\![.]\!] : \mathsf{WellTypClSpecVal} \to \mathsf{SemVal} \qquad [\![\texttt{nil}]\!] \triangleq [\,] \qquad [\![\pi \cdot \pi']\!] \triangleq [\![\pi]\!] \sqcup [\![\pi']\!]$$

Semantics of Formulas. We now state the semantics of formulas introduced to deal with reentrant locks:

$$\begin{array}{lcll}
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s & \models & \texttt{Lockset}(\pi) & \text{iff } \mathcal{L}(o) = [\![\pi]\!] \text{ for some } o \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s & \models & \pi \texttt{ contains } e & \text{iff } [\![e]\!]^h_s \in [\![\pi]\!] \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s & \models & e.\texttt{fresh} & \text{iff } [\![e]\!]^h_s \in \mathcal{F} \\
\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s & \models & e.\texttt{initialized} & \text{iff } [\![e]\!]^h_s \in \mathcal{I}
\end{array}$$

These clauses are self-explanatory, except perhaps the existential quantification in the clause for $\texttt{Lockset}(\pi)$. Intuitively, this clause says that there exists a thread identifier o in $\mathcal{L}$'s domain such that π denotes the current lockset associated with o.

Axioms. We axiomatize bag membership:

$$\Gamma; v \vdash \texttt{!(nil contains } e\texttt{)} \qquad \Gamma; v \vdash \texttt{(}\pi \cdot \pi'\texttt{) contains } e \texttt{ *-* (}e \texttt{ == } \pi \texttt{ | } \pi' \texttt{ contains } e\texttt{)}$$

To show equalities between specification values (formula e == π above), we axiomatize equality of specification values:

$$\Gamma; v \vdash \pi \texttt{ == } \pi \qquad \Gamma; v \vdash \pi \texttt{ == } \pi' \;\Rightarrow\; \Gamma; v \vdash \pi' \texttt{ == } \pi$$
$$\Gamma; v \vdash \pi \texttt{ == } \pi' \texttt{ \& } \pi' \texttt{ == } \pi'' \;\Rightarrow\; \Gamma; v \vdash \pi \texttt{ == } \pi''$$

In addition, we lift bag equality ($\simeq$) to our proof theory:

$$\pi \simeq \pi' \;\Rightarrow\; \Gamma; v \vdash \pi \texttt{ == } \pi'$$

Finally, we update Section 3.2.6's (Copyable) axiom about copyability of formulas:

$$G \in \{e, \texttt{ } \pi \texttt{ contains } e,\ e\texttt{.initialized} \texttt{ }\} \;\Rightarrow\; \Gamma; v \vdash \texttt{(}F \texttt{ \& } G\texttt{) -* (}F \texttt{ * } G\texttt{)} \qquad \text{(Copyable)}$$

5.4 Hoare Triples

In this section, we modify the Hoare triple (New) for allocating new objects and we show Hoare triples for the new commands of our language.

We modify rule (New) so that it emits the `fresh` predicate in its postcondition:

$$\frac{C\texttt{<}\bar{T}\,\bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma; v \vdash \begin{array}{c}\{\texttt{true}\} \\ \ell \texttt{ = new } C\texttt{<}\bar{\pi}\texttt{>} \\ \{\ell\texttt{.init} \texttt{ * } C \texttt{ classof } \ell \texttt{ * } \circledast_{\Gamma(u) <: \texttt{Object}} \ell \texttt{ != } u \texttt{ * } \ell\texttt{.fresh}\}\end{array}} \quad \text{(New)}$$

In addition to the usual `init` and `classof` predicates, (New)'s postcondition records that the newly created object is distinct from all other objects that are in scope. This postcondition is usually omitted in separation logic, because separation logic gets around explicit reasoning about the absence of aliasing. Unfortunately, we cannot entirely avoid this kind of reasoning when establishing the precondition for the rule (Lock) below, which requires that the lock is *not* already held by the current thread.

The specification command π`.commit` triggers π's transition from the `fresh` to the `initialized` state, provided π's resource invariant is established:

$$\frac{\Gamma \vdash \pi, \pi' : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\texttt{Lockset(}\pi'\texttt{) * }\pi\texttt{.inv * }\pi\texttt{.fresh}\} \\ \pi\texttt{.commit} \\ \{\texttt{Lockset(}\pi'\texttt{) * !(}\pi' \texttt{ contains } \pi\texttt{) * }\pi\texttt{.initialized}\}\end{array}} \quad \text{(Commit)}$$

Intuitively, the fact that π`.inv` appears in (Commit)'s precondition but does not appear in (Commit)'s postcondition indicates that after being committed, lock π begins to *guard* its resource invariant: the resource invariant π`.inv` has been given to lock π and π`.inv` is not available anymore to the executing thread. Furthermore, because π`.fresh` only holds if π `!=null`, this rule ensures that only non-null locks can become initialized.

The rule (Commit) ensures that monitor invariants cannot mention `Lockset` predicates. This is important because `Lockset` predicates are interpreted w.r.t. to the current

thread: having `Lockset` predicates inside monitor invariants does not make sense. The fact that monitor invariants cannot mention `Lockset` predicates is enforced by (Commit)'s precondition: because it mentions both a `Lockset` predicate and the lock's monitor invariant `inv`, `inv` cannot include a `Lockset` predicate. This follows from the semantics of the `Lockset` predicate and the semantics of the `*` operator: two `Lockset` predicates cannot be `*`-conjoined. Hence, if `inv` includes a `Lockset` predicate, (Commit)'s precondition cannot be established.

There are two rules each for locking and unlocking, depending on whether or not the `lock`/`unlock` is associated with an initial entry or a reentry.

First, we present the two rules for locking:

$$\frac{\Gamma \vdash u, \pi : \texttt{Object}, \texttt{lockset}}{\begin{array}{c}\Gamma; v \vdash \{\texttt{Lockset}(\pi) * !(\pi \texttt{ contains } u) * u.\texttt{initialized}\} \\ u.\texttt{lock}() \\ \{\texttt{Lockset}(u \cdot \pi) * u.\texttt{inv}\}\end{array}} \text{ (Lock)}$$

$$\frac{\Gamma \vdash u.\pi : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset}(u \cdot \pi)\} u.\texttt{lock}() \{\texttt{Lockset}(u \cdot u \cdot \pi)\}} \text{ (Re-Lock)}$$

The rule (Lock) applies when lock u is acquired non-reentrantly, as expressed by the precondition $\texttt{Lockset}(\pi) * !(\pi \texttt{ contains } u)$. The precondition $u.\texttt{initialized}$ makes sure that (1) threads only acquire locks whose resource invariant is initialized, and (2) no null-error can happen (because initialized values are non-null). The postcondition adds u to the current thread's lockset, and assumes u's resource invariant. The resource invariant obtained is $u.\texttt{inv}$ (without `@` selector). In proofs, the "visible" resource invariant is opened at u's static type using axioms (Open/Close) and (Dynamic Type) (these axioms are visible on page 33).

Proving (Lock)'s precondition requires reasoning about aliases because one has to prove $!(\pi \texttt{ contains } u)$. In practice, this assertion is proven by showing that u is different from all elements of lockset π. Such a reasoning is a form of alias analysis. On one hand this is unfortunate, because separation logic's power comes from the fact that it does not need to reason about aliases. On the other hand, this seems unavoidable. Whether this is problematic in practice needs to be investigated on large case studies. In the examples we study in this thesis (see Section 5.6), this precondition is a problem in one example out of three (the lock coupling example).

The rule (Re-Lock) applies when a lock is acquired reentrantly. The precondition of (Re-Lock), contrary to (Lock), does not require $u.\texttt{initialized}$, because this follows from $\texttt{Lockset}(u \cdot \pi)$ (locksets contain only initialized values).

To provide more useful feedback to programmers, we present a derived rule of (Re-Lock) that could be used in program checkers instead of (Re-Lock).

$$\frac{\Gamma \vdash u, \pi : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\texttt{Lockset}(u \cdot \pi) * u.\texttt{inv}\} \\ u.\texttt{lock}() \\ \{\texttt{Lockset}(u \cdot u \cdot \pi) * u.\texttt{inv}\}\end{array}} \text{ (Re-Lock-Accurate)}$$

Figure 5.1 shows method `syncCallToMth` that illustrates our point. To verify method `syncCallToMth`, a possible strategy for a program checker would be to do a case split on `(!S contains this | S contains this)` when reaching `lock()`. Now, consider the

```
requires Initialized * Lockset(S); ensures ...;
void syncCallToMth(){
   lock();
     mth(); // requires inv
   unlock();
}
```

Figure 5.1: Example showing (Re-Lock-Accurate)'s usefulness

case where `S contains this` holds (i.e., `this` is acquired reentrantly). When checking the program above, a program checker that uses (Re-Lock-Accurate) would fail at the call to `lock()` (because (Re-Lock-Accurate)'s precondition requires u.`inv`), while a program checker that uses (Re-Lock) would fail at the call to `mth()`. Because, the specification mistake concerns the usage of locks, not the call to `mth()`, using rule (Re-Lock-Accurate) would provide a more accurate feedback to the programmer than rule (Re-Lock). Generally speaking, rule (Re-Lock) is convenient for the theory but rule (Re-Lock-Accurate) is convenient in practice. In another words, upon lock reentry on a given lock o, one expects o's resource invariant to hold: this is what rule (Re-Lock-Accurate) enforces.

Second, we present the two rules for unlocking:

$$\frac{\Gamma \vdash u, \pi : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset}(u \cdot u \cdot \pi)\} u.\texttt{unlock}()\{\texttt{Lockset}(u \cdot \pi)\}} \text{ (Re-Unlock)}$$

$$\frac{\Gamma \vdash u, \pi : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset}(u \cdot \pi) * u.\texttt{inv}\} u.\texttt{unlock}()\{\texttt{Lockset}(\pi)\}} \text{ (Unlock)}$$

The rule (Re-Unlock) applies when u's current reentrancy level is at least 2 and (Unlock) applies when u's resource invariant holds in the precondition.

Other Hoare Rules that Do Not Work. One might wish to avoid the disequalities in (New)'s postcondition. Several approaches for this come to mind. First, one could drop the disequalities in (New)'s postcondition, and rely on (Commit)'s postcondition `!(`π' `contains` π`)` to establish (Lock)'s precondition. While this would be sound, in general it is too weak, as we are unable to lock π if we first lock some other object x (because from `!(`π' `contains` π`)` we cannot derive `!(`$x \cdot \pi'$ `contains` π`)` unless we know π `!=` x). Second, the `Lockset` predicate could be abandoned altogether, using a predicate π.`Held(`n`)` instead, that specifies that the current thread holds lock π with reentrancy level n. In particular, π.`Held(0)` means that the current thread does not hold π's lock at all. We could reformulate the rules for locking and unlocking using the `Held`-predicate, and introduce ℓ.`Held(0)` as the postcondition of (New), replacing the disequalities. However, this approach does not work, because it grants only the object creator permission to lock the created object! While it is conceivable that a clever program logic could somehow introduce π.`Held(0)`-predicates in other ways (besides introducing it in the postcondition of (New)), we have not been able to come up with a workable solution along these lines.

Wait and notify. Recall that in Section 5.2, we added methods `wait` and `notify` in class `Object` without specifying their contracts. Now we specify those as shown in Figure 5.2 on page 78.

```
class Object{
 pred inv = true;
 requires Lockset(S) * S contains this * inv;
 ensures  Lockset(S) * inv;
 final void wait();
 requires Lockset(S) * S contains this;
 ensures  Lockset(S);
 final void notify();
}
```

Figure 5.2: Class `Object`

The preconditions for `wait` and `notify` require that the receiver is locked. These requirements statically prevent `IllegalMonitorStateExceptions`, which are the runtime exceptions that Java throws when o.`wait()` or o.`notify()` are called without holding o's lock. The postcondition of o.`wait()` ensures o.`inv`, because o is locked just before o.`wait()` terminates.

Auxiliary Syntax. Recall that in Section 5.2, we added two new head commands `waiting` and `resume` to represent waiting states. The Hoare rules for these commands are as follows:

$$\frac{\Gamma \vdash \pi, o : \texttt{lockset}, \texttt{Object}}{\Gamma; r \vdash \begin{array}{c}\{\texttt{Lockset}(\pi) * o.\texttt{initialized}\} \\ o.\texttt{waiting}(n) \\ \{\texttt{Lockset}(\pi) * o.\texttt{initialized}\}\end{array}} \text{(Waiting)}$$

$$\frac{\Gamma \vdash o, \pi : \texttt{Object}, \texttt{lockset}}{\Gamma; r \vdash \begin{array}{c}\{\texttt{Lockset}(\pi) * o.\texttt{initialized}\} \\ o.\texttt{resume}(n) \\ \{\texttt{Lockset}(o^n \cdot \pi) * o.\texttt{inv}\}\end{array}} \text{(Resume)}$$

In (Resume), o^n denotes the multiset with n occurrences of o. More precisely: $o^0 = \texttt{nil}$ and $o^n = o \cdot o^{n-1}$ if $n \geq 1$. Of course, the rules (Waiting) and (Resume) are never used in source code verification, because source programs do not contain the auxiliary syntax. Instead, the rules (Waiting) and (Resume) are used to state and prove the preservation theorem.

5.5 Verified Programs

In this section, we show how to modify class `Thread` and top level rules of Chapter 4's verification system to handle reentrant locks.

The `Thread` *class.* To handle reentrant locks, we modify class `Thread`'s method contracts as shown in Figure 5.3. Intuitively, we forbid `fork` and `join`'s contracts (i.e., `preFork` and `postJoin`) to depend on the caller's lockset. This would not make sense since `Lockset` predicates are interpreted w.r.t. to the current thread. Obviously, a thread calling `fork` differs from the newly created thread, while a thread calling `join` differs from the joined

thread. We forbid `fork` and `join`'s contracts to depend on the caller's lockset by (1) adding `Lockset(S)` in `fork`'s precondition: because callers of `fork` have to establish `fork`'s precondition, this forbids `preFork` to depend on a `Lockset` predicate (recall that two `Lockset` predicates cannot be `*`-combined) and (2) by adding `Lockset(S)` in `run`'s postcondition: this forbids `postJoin` to depend on a `Lockset` predicate:

```
class Thread ext Object{
  pred preFork = true;
  group postJoin<perm p> = true;
  requires Lockset(S) * preFork; ensures Lockset(S);
  final void fork();
  requires Join(this,p); ensures postJoin<p>;
  final void join();
  final requires Lockset(nil) * preFork;
        ensures (ex Lockset S)(Lockset(S)) * postJoin<1>;
  void run() { null }
}
```

Figure 5.3: Class `Thread`

Top Level Rules. We need to update Section 4.5's rules for runtime states to account for reentrant locks.

There are two changes to rule (Thread): (1) premise $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) = \{o\}$ is added to ensure that a thread's resource only tracks the locks held by this thread and (2) the thread's postcondition is modified to reflect the change in `join`'s postcondition in class `Thread`.

(Thread)

$$\frac{\begin{array}{c}\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!] \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \\ \mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset \quad \mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) = \{o\} \quad \Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma] \\ \Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{\texttt{(ex lockset}\,S\texttt{)(Lockset(}S\texttt{))} * fr \cdot o.\texttt{postJoin<1>}\}\end{array}}{\mathcal{R} \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c) : \diamond}$$

We define the set $\mathsf{ready}(\mathcal{R})$ of all initialized objects whose locks are not held, and the function conc that maps abstract lock tables to concrete lock tables:

$$\mathsf{ready}(\mathcal{R}) \triangleq \mathcal{R}_{\mathsf{init}} \setminus \{\, o \mid (\exists p)(o \in \mathcal{L}(p)) \,\}$$

$$\mathsf{conc}(\mathcal{L})(o) \triangleq \begin{cases} (p, \mathcal{L}(p)(o)) & \text{iff } o \in \mathcal{L}(p) \\ \mathsf{free} & \text{otherwise} \end{cases}$$

In conc's definition, we let $\mathcal{L}(p)(o)$ stand for the multiplicity of o in $\mathcal{L}(p)$. Note that conc is well-defined, by axiom (b) for resources (see page 73). The new rule for states ensures that there exists a resource $\mathcal{R}$ to satisfy the thread pool ts and a resource $\mathcal{R}'$ to satisfy the resource invariants of the locks that are ready to be acquired. In addition,

function conc relates the program's lock table to the top level resource's abstract lock table:

$$\frac{h = (\mathcal{R} * \mathcal{R}')_{\mathsf{hp}} \qquad l = \mathsf{conc}(\mathcal{R}_{\mathsf{lock}}) \qquad \mathcal{R} \vdash ts : \diamond \qquad \mathcal{R}\#\mathcal{R}' \quad \mathcal{R}'_{\mathsf{lock}} = \emptyset \quad \mathsf{fst}(\mathcal{R}'_{\mathsf{hp}}) \subseteq \mathsf{fst}(h) = \Gamma \quad \Gamma \vdash \mathcal{E}; \mathcal{R}'; \emptyset \models \circledast_{o \in \mathsf{ready}(\mathcal{R})} o.\mathtt{inv}}{\langle h, l, ts \rangle : \diamond} \quad \text{(State)}$$

Like in Section 4.5, we have shown the preservation Theorem 3 (Section 6.2.2's proof is extended in Section 6.3) and we have shown that verified programs satisfy the following properties: null error freeness, partial correctness, and data race freeness.

5.6 Examples of Reasoning with Reentrant Locks

In this section, we show examples of reasoning with our verification system. We provide three examples: Section 5.6.1's class `Set` shows a typical use of reentrant locks as it often occurs in the Java library, Section 5.6.2 shows an implementation of the *worker thread* design pattern [76, §4.1.4], and Section 5.6.3 shows an advanced lock coupling example.

5.6.1 A Typical Use of Reentrant Locks: class `Set`

In the Java library, lock reentrancy is useful because container classes often feature client methods that are also helper methods. This happens if there is (1) a method which synchronizes on the receiver and is meant to be called by clients, but (2) this method can also be called by other methods of the same class. Because the other methods can also be synchronized on the receiver, lock reentrancy avoids to duplicate method implementation in two versions: a synchronized one (to be called by clients) and a lock free one (to be called by other methods of the class).

We present an example of the behavior described above for a class `Set` that represents mathematical sets. Internally, class `Set` is backed up by a list. Class `Set` contains a method `has` that should be used by clients to check if some element belongs to the receiver set. In addition, class `Set` contains method `add` which adds an element to the receiver set if this element is not already present. Both method `has` and method `add` lock the receiver set. Hence, as method `add` calls `has`, reentrant locks are crucial for class `Set`'s implementation.

First, we provide class `List` that backs up class `Set`. Class `List` is a *shallow* container: lists do not have permission to access their values. Values must be accessed by synchronizing on them. That is why lists ensure that they only contain initialized values (see predicate `state`'s implementation):

```
class List extends Object{
 Object element;
 List next;
 pred state = PointsTo(element,1,v) * v.initialized *
              PointsTo(next,1,n) * n.state;
 requires init * o.initialized; ensures state@List;
 void init(Object o, List n){ element = o; next = n; }
 requires state; ensures state;
```

```
 bool has(Object o){
  bool result;
  if(element == o){ result = true; }
  else{
   if(next != null){ result = next.has(o); }
   else{ result = false; }
  }
  result;
 }
 requires state * o.initialized; ensures state;
 void add(Object o){
  List l = new List;
  l.init(o,this);
 }
}
```

We explain the meaning of class `List`'s predicate `state`. Predicate `state` gives access to field `next` of the list's first node (see `PointsTo(next,1,n)`) and to all `next` fields of subsequent nodes (because `state` is recursive, see `n.state`). In addition, predicate `state` (1) provides references to the values stored in the list (see `PointsTo(element,1,v)` and `n.state`) and (2) ensures that values are initialized (see `v.initialized` and `n.state`). It is crucial to ensure that values inside lists are initialized because predicate `state` does not give access to the values, it only provides references.

Second, we provide class `Set`. Class `Set` ensures that an object cannot appear twice in the underlying list. For simplicity, we identify two objects if they have the same address in the heap (i.e., we use Java's `==`)[1]:

```
class Set extends Object{
 List rep;
 pred inv = PointsTo(rep,1,r) * r.state;
 requires Lockset(S) * init * fresh * Set classof this * o.initialized;
 ensures  Lockset(S) * !(S contains this) * initialized;
 void init(Object o){
  rep = new List;
  rep.init(o,null);
  commit;
 }
 requires Lockset(S) * (S contains this -* inv) * initialized;
 ensures  Lockset(S) * (S contains this -* inv);
 bool has(Object o){
  lock();
   bool result = rep.has(o);
  unlock();
  result;
```

[1] Alternatively, we could put Java's `equals` in class `Object` and use it here.

```
 }
 requires Lockset(S) * !(S contains this) * initialized * o.initialized;
 ensures  Lockset(S) * !(S contains this);
 void add(Object o){
  lock();
   if(!has(o)){ // lock-reentrant call
    rep.add(o);
   }
  unlock();
 }
}
```

Remarks.

- The resource invariant of a `Set` consists of (1) the field `rep` and (2) the list pointed to by the field `rep`. This is specified in predicate `inv`'s implementation.
- A `Set` *owns* its underlying list `rep`: while the receiver set is locked when clients call `has` or `add`, the underlying list is never locked. Access rights to the underlying list are packed into the resource invariant of the set (see `inv`'s definition). As a result, lists do not need to be initialized (no `commit` statement in class `List`).
- Elements of sets should be accessed by synchronizing on them. Although there is no `get` method in class `Set`'s implementation, we make sure that elements of sets are `initialized` (see `state`'s implementation in class `List` and `o.initialized` in various contracts). Hence, a `get` method would have `result.initialized` as a postcondition, allowing clients to lock returned elements.
- Method `init` both (1) initializes field `rep` and (2) initializes the set's resource invariant (with the `commit` command). Point (2) is formalized by having `fresh` in `init`'s precondition and having `initialized` in `init`'s postcondition. In addition, `init`'s precondition includes `Set classof this`. This is required to verify that `commit` is sound i.e., that the monitor invariant is established before `commit` (see `init`'s proof outline below).
- Contract of method `has` in class `Set` allows lock-reentrant calls. If a lock-reentrant call is performed, however, `inv` is required (as expressed by `(S contains this -* inv)`). Method `add` in class `Set` could be specified similarly.
- A simpler implementation of method `add` in class `Set` would call `has` on the underlying list. In this way, the lock-reentrant call would be avoided. However, our implementation is safer: if method `has` is overridden in subclasses of `Set` (but not method `add`), our implementation is still correct; while the simpler implementation could exhibit unexpected behaviors.

Classes `List` and `Set` have been verified with our system. Below, we verify method `init` of class `Set`:

```
{ Lockset(S) * init * fresh * Set classof this * o.initialized }
((Known Type) axiom and abbreviation of Set classof this by F)
{ Lockset(S) * init@Set * fresh * F * o.initialized }
((Open/Close) axiom)
{ Lockset(S) * PointsTo(rep,1,null) * fresh * F * o.initialized }
```

```
rep = new List;
 (Weakening)
 { Lockset(S) * PointsTo(rep,1,r) * fresh * F * r.init * o.initialized }
rep.init(o,null);
 { Lockset(S) * PointsTo(rep,1,r) * fresh * F * r.state@List }
 ((Known Type) axiom on r)
 { Lockset(S) * PointsTo(rep,1,r) * fresh * F * r.state }
 ((Known Type) axiom on this)
 { Lockset(S) * inv * fresh }
commit;
 { Lockset(S) * !(S contains this) * initialized }
```

Now, we verify method `add` of class `Set`:

```
 { Lockset(S) * !(S contains this) * initialized * o.initialized }
lock();
 { Lockset(S.this) * inv * o.initialized }
 (has's precondition is satisfied: S.this contains this and inv is present)
if(has(o))
   (Case 1: the conditional holds)
   { Lockset(S.this) * inv * o.initialized }
   ((Dynamic Type) axiom and modus ponens)
   { Lockset(S.this) * inv@Set * (inv@Set -* inv) * o.initialized }
   ((Open/Close) axiom)
   { Lockset(S.this) * PointsTo(rep,1,r) * r.state * (inv@Set -* inv) *
     o.initialized }
 rep.add(o);
   { Lockset(S.this) * PointsTo(rep,1,r) * r.state * (inv@Set -* inv) }
   ((Open/Close) axiom)
   { Lockset(S.this) * inv@Set * (inv@Set -* inv) }
   (Modus ponens)
   { Lockset(S.this) * inv }
 unlock();
   { Lockset(S) }

   (Case 2: the conditional does not hold)
   { Lockset(S.this) * inv * o.initialized }
 unlock();
   (Weakening)
   { Lockset(S) }
```

Discussion. Class `Set` exemplifies a typical use of lock reentrancy in the Java library. We believe that our verification system fits well to verify such classes. In addition, this example shows how our system supports programs that include objects that must be locked before access and objects that are accessed without synchronization. Importantly, the addition of locks does not force programmers to indicate `Lockset` predicates everywhere in contracts: class `List` which backs up class `Set` does not mention any `Lockset` predicates.

5.6.2 The Worker Thread Design Pattern

The *worker thread* pattern [76, §4.1.4] consists of one thread (or more, in which case we say there is a *pool* of worker threads) that performs many unrelated tasks in succession. Client threads simply send tasks that must be executed to the worker thread. The worker thread keeps pending tasks in some container and executes the task with the highest priority first. This design pattern is of crucial importance in a variety of applications, including cellular phones [37] and GUIs. Below, we show how to specify and verify an implementation of this design pattern in our system.

Tasks are represented with an interface with only one method: `process`. Because tasks should be accessed by synchronizing on them, `process`'s pre- and postcondition is just `initialized`:

```
interface Task{
 requires initialized; ensures initialized;
 void process;
}
```

To store tasks in the worker thread, we use a generic `Container` that has methods `push` and `pop`. In practice, `Container` could be implemented by a stack, a FIFO queue etc. Method `pop`'s postcondition uses the special variable `result` to refer to the object returned:

```
interface Container{
 pred state;
 requires init; ensures state@Container;
 void init();
 requires state * t.initialized; ensures state;
 void push(Task t);
 requires state; ensures state * (result==null | result.initialized);
 Task pop();
}
```

We now have all the machinery to implement a worker thread. Method `run` implements the behavior of the worker thread. First, the worker thread looks up if there is a task pending by calling `pop()`. Second, two cases are possible: (1) if there is a task pending (`pop` returned a non-null task), the task is processed and the worker thread immediately looks up a new task or (2) if there is no task pending, the worker thread waits until a client notifies it that a new task has been added. We highlight that the task is processed without holding the worker thread's lock to increase parallelism. Client threads send tasks by calling method `add`. Because we do not have loops, we mimic a `while(true)` loop with recursive calls to `run`. As worker threads are supposed to run forever and because we only consider partial correctness, this does not harm soundness.

```
class WorkerThread extends Thread{
 Container c;
 pred inv = PointsTo(c,1,v) * v.state;
```

```
 pred preFork = initialized;
 group postJoin<perm p> = initialized;
 requires Lockset(S) * fresh * init;
 ensures  Lockset(S) * !(S contains this) * initialized ;
 void init(){
  c = new Container();
  c.init();
  commit;
 }
 requires Lockset(nil) * preFork;
 ensures  (ex Lockset S)(Lockset(S)) * postJoin<1>;
 void run(){
  lock();
   Task t = c.pop();
  unlock();
  if(t!=null){ t.process(); run(); }
  else{
   lock();
    wait();
   unlock();
   run();
  }
 }
 requires Lockset(S) * t.initialized; ensures Lockset(S);
 void add(Task t){
  lock();
   c.push(t);
   notify();
  unlock();
 }
}
```

We verify method `init` of class `WorkerThread`:

```
 { Lockset(S) * fresh * init }
```
((Dynamic Type) axiom, modus ponens, and weakening)
```
 { Lockset(S) * fresh * init@WorkerThread }
```
((Open/Close) axiom)
```
 { Lockset(S) * fresh * PointsTo(c,1,null) * Join(this,1) }
```
(Weakening)
```
 { Lockset(S) * fresh * PointsTo(c,1,null) }
c = new Container();
```
(Weakening)
```
 { Lockset(S) * fresh * PointsTo(c,1,v) * v.init * Container classof v }
c.init();
 { Lockset(S) * fresh * PointsTo(c,1,v) * v.state@Container *
   Container classof v }
```

((Known Type) axiom)
{ Lockset(S) * fresh * PointsTo(c,1,v) * v.state }
((Open/Close) axiom)
{ Lockset(S) * fresh * inv }
commit;
{ Lockset(S) * (!S contains this) * initialized }

We verify method run of class WorkerThread:

{ Lockset(nil) * preFork }
((Dynamic Type) axiom and modus ponens)
{ Lockset(nil) * preFork@WorkerThread * (preFork@WorkerThread -* preFork) }
((Open/Close) axiom and abbreviating preFork@WorkerThread -* preFork by F)
{ Lockset(nil) * initialized * F }
((Copyable) on initialized)
lock();
{ Lockset(this) * initialized * inv * F }
((Dynamic Type) axiom and modus ponens)
{ Lockset(this) * initialized * inv@WorkerThread *
(inv@WorkerThread -* inv) * F }
((Open/Close) axiom and abbreviating inv@WorkerThread -* inv by G)
{ Lockset(this) * initialized * PointsTo(c,1,v) * v.state * F * G }
Task t = c.pop();
{ Lockset(this) * initialized * PointsTo(c,1,v) * v.state *
(t==null | t.initialized) * F * G }
((Open/Close) and modus ponens)
{ Lockset(this) * initialized * inv * (t==null | t.initialized) * F }
unlock();
{ Lockset(nil) * initialized * (t==null | t.initialized) * F }
if(t!=null)
(Case 1: the conditional holds)
{ Lockset(nil) * initialized * t.initialized * F * t!=null }
t.process();
{ Lockset(nil) * initialized * t.initialized * F }
((Open/Close) and weakening)
{ Lockset(nil) * preFork@WorkerThread * F }
(Modus ponens)
{ Lockset(nil) * preFork }
run();

(Case 2: the conditional does not hold)
{ Lockset(nil) * initialized * t==null * F }
((Copyable) on initialized and weakening)
lock();
{ Lockset(this) * initialized * inv * F }
wait();
{ Lockset(this) * initialized * inv * F }

```
unlock();
  { Lockset(nil) * initialized * F }
  ((Open/Close))
  { Lockset(nil) * preFork@WorkerThread * F }
  (Modus ponens)
  { Lockset(nil) * preFork }
run();
```

Discussion. This design pattern does not raise any problem for our verification system: we are able to verify method contracts in a straightforward way. The reason is that, in method `run` of class `WorkerThread`, the lockset is always precisely determined. Either it is empty, or it only contains the worker thread itself, or it only contains the task being processed (during the call to `process`). For this reason, our system handles this example well. In the next section, we show an example where locksets raise real problems because they cannot be determined precisely. In this situation, ownership reasoning is needed.

5.6.3 Lock Coupling

In this section, we illustrate how our verification system handles lock coupling. We use the following convenient abbreviations:

$$\pi\texttt{.locked}(\pi') \triangleq \texttt{Lockset}(\pi \cdot \pi') \quad \pi\texttt{.unlocked}(\pi') \triangleq \texttt{Lockset}(\pi') * \texttt{!}(\pi' \texttt{ contains } \pi)$$

Suppose we want to implement a sorted linked list with repetitions. For simplicity, assume that the list has only two methods: `insert()` and `size()`. The former inserts an integer into the list, and the latter returns the current size of the list. To support a constant-time `size()`-method, each node stores the size of its tail in a `count`-field. Each node n maintains the invariant n`.count == `n`.next.count + 1`.

In order to allow multiple threads inserting simultaneously, we want to avoid using a single lock for the whole list. We have to be careful, though: a naive locking policy that simply locks one node at a time would be unsafe, because several threads trying to simultaneously insert the same integer can cause a semantic data race, so that some integers get lost and the `count`-fields get out of sync with the list size. The lock coupling technique avoids this by simultaneously holding locks of two neighboring nodes at critical times.

Lock coupling has been used as an example by Gotsman et al. [51] for single-entrant locks. The additional problem with reentrant locks is that `insert()`'s precondition must require that none of the list nodes is in the lockset of the current thread. This is necessary to ensure that on method entry the current thread is capable of acquiring all nodes's resource invariants:

```
requires this.unlocked(S) * no list node is in S;
ensures  Lockset(S);
void insert(int x);
```

The question is how to formally represent the informal condition written in italic. Our solution makes use of class parameters. We require that nodes of a lock-coupled list are *statically owned* by the list object, i.e., they have type `Node<`o`>`, where o is the list object. Then we can approximate the above contract as follows:

```
requires this.unlocked(S) * no this-owned object is in S;
ensures  Lockset(S);
void insert(int x);
```

To express this formally, we define a marker interface, i.e., an interface with no content, for owned objects:

```
interface Owned<Object owner> { /* a marker interface */ }
```

Next we define an auxiliary predicate π.`traversable(`π'`)` (read as "if the current thread's lockset is π', then the aggregate owned by object π is traversable"). Concretely, this predicate says that no object owned by π is contained in π':

$$\pi\texttt{.traversable(}\pi'\texttt{)} \triangleq \texttt{(fa Object owner, Owned<owner> x)(!(}\pi'\texttt{ contains x) | owner != }\pi\texttt{)}$$

Note that in our definition of π.`traversable(`π'`)`, we quantify over a type parameter (namely the `owner`-parameter of the `Owned`-type). Here we are taking advantage of the fact that program logic and type system are inter-dependent.

Now, we can formally define an interface for sorted integer lists:

```
interface SortedIntList {
 pred inv<int c>; // c is the number of list nodes
 requires this.inv<c>; ensures this.inv<c> * result==c;
 int size();
 requires this.unlocked(s) * this.traversable(s); ensures Lockset(s);
 void insert(int x);
}
```

Figure 5.4 shows a tail-recursive lock-coupling implementation of `SortedIntList`. The auxiliary predicate n.`couple<`c, c'`>`, as defined in the `Node` class, holds in states where n.`count ==` c and n.`next.count ==` c'. Figure 5.4's implementation has been verified in our system.

But how can clients of lock-coupling lists establish `insert()`'s precondition? The answer is that client code needs to track the types of locks held by the current thread. For instance, if C is not a subclass of `Owned`, then `list.insert()`'s precondition is implied by the following assertion, which is satisfied when the current thread has locked only objects of types C and `Owned<`ℓ`>`.

```
list.unlocked(S) * ℓ!=list *
(fa Object z)(!(S contains z) | z instanceof C | z instanceof Owned<ℓ>)
```

Discussion. This example demonstrates that we can handle fine-grained concurrency despite the technical difficulties raised by lock reentrancy (i.e., `lock`'s precondition is harder to prove). However, we have to fall back on the type system to verify this example. Consequently, ownership becomes *static*: with our specifications, nodes cannot be transferred from a list to another, because the nodes's owner would have to change. As a result, our solution works for a limited set of programs and further work is needed to handle all uses of fine-grained concurrency. In addition, because we use that our type system and

```
class LockCouplingList implements SortedIntList{
     Node<this> head;
     pred inv<int c> = (ex Node<this> n)(
         PointsTo(head, 1, n) * n.initialized * PointsTo(n.count, 1/2, c) );
     requires this.inv<c>; ensures this.inv<c> * result==c;
     int size() { return head.count; }
     requires Lockset(S) * !(s contains this) * this.traversable(s);
     ensures  Lockset(S);
     void insert(int x) {
        lock(); Node<this> n = head;
        if (n!=null) {
           n.lock();
           if (x <= n.val) {
              n.unlock(); head = new Node<this>(x,head); head.commit; unlock();
           } else { unlock(); n.count++; n.insert(x); }
        } else { head = new Node<this>(x,null); unlock(); } } }

class Node<Object owner> implements Owned<owner>{
     int count; int val; Node<owner> next;
     public pred couple<int count_this, int count_next> =
        (ex Node<owner> n)(
              PointsTo(this.count, 1/2, count_this) * PointsTo(this.val, 1,int)
           * PointsTo(this.next, 1, n) * n!=this * n.initialized
           * ( n!=null -* PointsTo(n.count, 1/2, count_next) )
           * ( n==null -* count_this==1 ) );
     public pred inv<int c> = couple<c,c-1>;
     requires PointsTo(next.count, 1/2, c);
     ensures PointsTo(next.count, 1/2, c)
       * ( next!=null -* PointsTo(this.count, 1, c+1) )
       * ( next==null -* PointsTo(this.count, 1, 1) )
       * PointsTo(this.val, 1, val) * PointsTo(this.next, 1, next);
     Node(int val, Node<owner> next) {
        if (next!=null) { this.count = next.count+1; } else { this.count = 1; }
        this.val = val; this.next = next; }
     requires Lockset(this·S) * owner.traversable(s) * this.couple<c+1,c-1>;
     ensures Lockset(S);
     void insert(int x) {
        Node<owner> n = next;
        if (n!=null) {
           n.lock();
           if (x <= n.val) {
              n.unlock(); next = new Node<owner>(x,n); next.commit; unlock();
           } else { unlock(); n.count++; n.insert(x); }
        } else { next = new Node<owner>(x, null); unlock(); } } }
```

Figure 5.4: A lock-coupling list

our verification system are inter-dependent, automatically proving such a program would require a tool that combines reasoning about types and reasoning about separation logic formulas.

5.7 Related Work and Conclusion

Related Work. There are a number of similarities between our work and Gotsman et al. [51]'s work, for instance the treatment of initialization of dynamically created locks. Our `initialized` predicate corresponds to what Gotsman calls lock handles (with his lock handle parameters corresponding to our class parameters). Since Gotsman's language supports deallocation of locks, he scales lock handles by fractional permissions in order to keep track of sharing. This is not necessary in a garbage-collected language. In addition to single-entrant locks, Gotsman also treats thread joining which we covered in the previous chapter. The essential differences between Gotsman's and our paper are (1) that we treat reentrant locks, which are a different synchronization primitive than single-entrant locks, and (2) that we treat subclassing and extension of resource invariants in subclasses. Hobor et al.'s work [58] is very similar to [51].

Another related line of work is by Jacobs et al. [70] who extend the Boogie methodology for reasoning about object invariants [7] to a multithreaded Java-like language. While their system is based on classical logic (without operators like $*$ and $-\!\!*$), it includes built-in notions of ownership and access control. Their system deliberately enforces a certain programming discipline (like concurrent separation logic and our variant of it also do) rather than aiming for a complete program logic. The object life cycle imposed by their discipline is essentially identical to ours. For instance, their `shared` objects (objects that are shared between threads) directly correspond to our `initialized` objects (objects whose resource invariants are initialized). Their system prevents deadlocks, which our system does not. They achieve deadlock prevention by imposing a partial order on locks. As a consequence of their order-based deadlock prevention, their programming discipline statically prevents reentrancy, although it may not be too hard to relax this at the cost of additional complexity.

A different approach is pursued by Vafeiadis et al. [98, 44]. This work combines rely/guarantee with separation logic. On one hand, this is both powerful and flexible: fine-grained concurrent algorithms can be specified and verified. On the other hand, their verification system is more complex than ours. They do not treat reentrant locks.

In a more traditional approach, Ábráham, De Boer et al. [1, 39] apply assume-guarantee reasoning to a multithreaded Java-like language.

Conclusion. We showed a Java-like language with reentrant locks. Our key contributions are as follows: (1) we presented verification rules for reentrant locks, (2) we showed how resource invariants can be expressed (including inheritance) with general purpose abstract predicates, and (3) we described how the combination of an expressive type system and separation logic handles a challenging lock-coupling algorithm. We remark that we do not need to introduce new machinery to handle inheritance in combination with locks: abstract predicates suffice.

The material described in Chapters 2, 3, 4, and in this chapter defines a verification system for multithreaded Java. It covers the two main features of multithreaded Java programs: `fork` and `join` primitives (Chapters 3 and 4) and reentrant locks (this chapter). The next chapter shows that this verification system is sound.

Chapter 6

Soundness of Chapter 3 to Chapter 5's Verification System

In this chapter, we prove that the verification system described from Chapter 3 to 5 is sound. This is an important goal, because static checking is supposed to offer a high-level of confidence to verified programs. Consequently, it is crucial to show that the static checking algorithm itself is correct.

The proofs are separated for each chapter. In Section 6.1, we show the soundness of Chapter 3's verification system, in Section 6.2, we show the soundness of Chapter 4's verification system, and in Section 6.3, we show the soundness of Chapter 5's verification system. In each section, the main proof is the preservation result. This proof shows that Hoare rules (Sections 3.3, 4.4, and 5.4) for verification of programs are correct and rules for verified programs (Sections 3.5, 4.5, and 5.5) are correct.

6.1 Soundness of Chapter 3's Verification System

6.1.1 Properties

In this section, we collect standard lemmas that the system is designed to satisfy. These lemmas are used in the proof of Theorems 3, 4, and 5; and in later proofs.

Properties of the Typing Judgments

Lemma 2 (Substitutivity and Inverse Substitutivity for Subtyping)**.**

(a) *If* $T <: U$*, then* $T[\sigma] <: U[\sigma]$*.*
(b) *If* $T[\sigma] <: U$*, then* $U = U'[\sigma]$ *for some* U'*.*
(c) *If* $T[\sigma] <: U[\sigma]$*, then* $T <: U$*.*

Proof. All three parts by induction on the derivation of the subtyping judgment. The proof of part (c) uses part (b) to deal with the transitivity rule. □

Lemma 3 (Inverse Substitutivity for Values)**.** *If* $(\Gamma \vdash \sigma : \Gamma')$ *and* $(\Gamma[\sigma] \vdash v : T[\sigma])$*, then* $(\Gamma, \Gamma' \vdash v : T)$*.*

Proof. In case v is an integer, boolean or `null`, this is obvious. So suppose that v is an object identifier or a read-only variable. Then $v \in \mathsf{dom}(\Gamma)$ and $\Gamma[\sigma](v) <: T[\sigma]$. But then $\Gamma(v) <: T$, by Lemma 2. But then $(\Gamma, \Gamma' \vdash v : T)$. □

Properties of the Proof Theory

Lemma 4 (Substitutivity)**.** *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v; \bar{F} \vdash G)$*, then* $(\Gamma; v; \bar{F})[\bar{\pi}/\bar{x}] \vdash G[\bar{\pi}/\bar{x}]$.

Proof. By induction on the derivation of $(\Gamma, \bar{x} : \bar{T}; v; \bar{F} \vdash G)$. □

Properties of Hoare Triples

Lemma 5 (Substitutivity)**.**

(a) *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v \vdash \{F\}hc\{G\})$*, then* $((\Gamma; v)[\bar{\pi}/\bar{x}] \vdash \{F[\bar{\pi}/\bar{x}]\}hc[\bar{\pi}/\bar{x}]\{G[\bar{\pi}/\bar{x}]\})$.

(b) *If* $(\Gamma[\bar{\pi}/\bar{x}] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{x}])$ *and* $(\Gamma, \bar{x} : \bar{T}; v \vdash \{F\}c : U\{G\})$*, then* $((\Gamma; v)[\bar{\pi}/\bar{x}] \vdash \{F[\bar{\pi}/\bar{x}]\}c[\bar{\pi}/\bar{x}] : U[\bar{\pi}/\bar{x}]\{G[\bar{\pi}/\bar{x}]\})$.

Proof. For hc by inspection of the last rule. For c by induction on the structure of c. □

Lemma 6 (Logical Consequence)**.** *If* $(\Gamma; v; F \vdash F')$ *and* $(\Gamma; v \vdash \{F'\}c : T\{G\})$*, then* $(\Gamma; v \vdash \{F\}c : T\{G\})$.

Proof. By induction on the structure of c. □

Lemma 7 (Derived Rule for Bind)**.** *If* $(\Gamma; o \vdash \{F\}c : T\{\texttt{(ex}\ T\ \alpha\texttt{)(}G\texttt{)}\})$*,* $T <: \Gamma(\ell)$ *and* $(\Gamma; p \vdash \{\texttt{(ex}\ T\ \alpha\texttt{)(}\alpha\ \texttt{==}\ \ell\ \texttt{*}\ G\texttt{)}\}c' : U\{H\})$*, then* $(\Gamma; o \vdash \{F\}\ell \leftarrow c;\ c' : U\{H\})$.

Proof. By induction on the structure of c. □

Properties of the Semantics

Lemma 8 (Resource Monotonicity)**.** *If* $(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F)$*,* $\mathcal{R} \leq \mathcal{R}'$*,* $\Gamma \subseteq_{\mathsf{hp}} \Gamma'$ *and* $(\Gamma'_{\mathsf{hp}} \vdash \mathcal{R}'_{\mathsf{hp}} : \diamond)$*, then* $(\Gamma' \vdash \mathcal{E}; \mathcal{R}'; s \models F)$.

Proof. By induction on the structure of F. For the cases where $F = o.P\texttt{<}\bar{\pi}\texttt{>}$ or $F = o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$, one uses that predicate environments are monotone with respect to resources, by axiom (a).

This proof has been mechanically checked for a smaller specification language (without abstract predicates and quantifiers) [62]. □

Lemma 9 (Stack Invariance)**.**

(a) *If* $s_{|\mathsf{fv}(e)} = s'_{|\mathsf{fv}(e)}$ *and* $[\![e]\!]^h_s = \mu$*, then* $[\![e]\!]^h_{s'} = \mu$.

(b) *If* $s_{|\mathsf{fv}(F)} = s'_{|\mathsf{fv}(F)}$*,* $\Gamma_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$*,* $\Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)}$*,* $(\Gamma' \vdash s' : \diamond)$ *and* $(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F)$*, then* $(\Gamma' \vdash \mathcal{E}; \mathcal{R}; s' \models F)$.

Proof. Part (a) by induction on the structure of e. Part (b) by induction on the structure of F. The proof is very similar to the proof of Lemma 72 in [54]. □

Auxiliary Lemmas

We lift field update to resources:

$$\mathcal{R}[o.f \mapsto v] \triangleq (\mathcal{R}_{\mathsf{hp}}[o.f \mapsto v], \mathcal{R}_{\mathsf{perm}})$$

The following lemma is needed to prove soundness of the verification rule (Fld Set).

Lemma 10 (Field Update)**.** *Suppose* $\mathcal{R}_{\mathsf{perm}}(o, f) = 1$, $\_ \; T \; f \in \mathsf{fld}(\mathcal{R}_{\mathsf{hp}}(o)_1)$ *and* $\mathsf{fst}(\mathcal{R}_{\mathsf{hp}}) \vdash v : T$*. Then:*

(a) $\mathcal{R}[o.f \mapsto v] \in \mathsf{Resources}$
(b) *If* $\mathcal{R}\#\mathcal{R}'$ *and* $o \in \mathsf{dom}(\mathcal{R}'_{\mathsf{hp}})$,
then $\mathcal{R}[o.f \mapsto v]\#\mathcal{R}'$ *and* $\mathcal{R}[o.f \mapsto v] * \mathcal{R}' = (\mathcal{R} * \mathcal{R}')[o.f \mapsto v]$.

We define:

$$\begin{aligned}
\mathsf{initloc}(o)(p,k) &\triangleq \begin{cases} 1 & \text{if } p = o \\ 0 & \text{otherwise} \end{cases} \\
\mathsf{inithp}(\Gamma, o, T)(p) &\triangleq \begin{cases} (T, \mathsf{initStore}(T)) & \text{if } p = o \\ (\Gamma(o), \emptyset) & \text{if } p \in \mathsf{dom}(\Gamma) \setminus \{o\} \end{cases} \\
\mathsf{initrsc}(\Gamma, o, T) &\triangleq (\mathsf{inithp}(\Gamma, o, T), \mathsf{initloc}(o))
\end{aligned}$$

The following lemma is needed to prove soundness of the verification rule (New).

Lemma 11 (Initialization)**.**

(a) *If* $(\Gamma \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond)$, $o \notin \mathsf{dom}(\Gamma)$, $(\Gamma \vdash s : \diamond)$, $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$ *and* $C \preceq D$*, then* $(\Gamma, o : C\texttt{<}\bar{\pi}\texttt{>} \vdash \mathcal{E}; \mathsf{initrsc}(\Gamma, o, C\texttt{<}\bar{\pi}\texttt{>}); s \models o.\texttt{init@}D)$.
(b) *If* $(\Gamma \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond)$, $o \notin \mathsf{dom}(\Gamma)$, $(\Gamma \vdash s : \diamond)$, *and* $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$*, then* $(\Gamma, o : C\texttt{<}\bar{\pi}\texttt{>} \vdash \mathcal{E}; \mathsf{initrsc}(\Gamma, o, C\texttt{<}\bar{\pi}\texttt{>}); s \models o.\texttt{init})$.

Proof. Part (a) by induction on the subclassing order $\preceq$. Part (b) follows from part (a) because $o.\texttt{init}$ is equivalent to $o.\texttt{init@}C$, if C is o's dynamic class. □

6.1.2 Preservation

In this section, we prove the preservation theorem 3.

First, we observe that we can normalize Hoare proofs for head commands as follows:

Lemma 12 (Proof Normalization)**.** *If* $(\Gamma; v \vdash \{F\} hc \{G\})$ *is derivable, then it has a proof where every path to the proof goal ends in zero or more applications of (Consequence) and (Exists) preceded by exactly one application of (Frame), preceded by a rule that is not a structural rule (i.e., a rule different from (Frame), (Consequence), and (Exists)).*

Proof. We need to show that we can permute an application of (Frame) upwards, when preceded by (Consequence) or (Exists). These permutations are straightforward to show. By associativity of $*$ we can condense a sequence of several (Frame) applications into a single (Frame) application. By neutrality of `true`, we can expand zero (Frame) applications into one (Frame) application. □

Proof of Theorem 3 (Preservation). *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$*, then* $(st' : \diamond)$.

Proof.

(1) $ct : \diamond$ assumption
(2) $st : \diamond$ assumption
(3) $st \rightarrow st'$ assumption

An inspection of the reduction rules shows that st is of the following form.

(4) $st = \langle h, c, s \rangle$

The proof of $(st : \diamond)$ consists of an application of (State). This application has the following premises:

(5) $\Gamma \vdash \sigma : \Gamma'$
(6) $\mathsf{dom}(\Gamma') \cap \mathsf{cfv}(c) = \emptyset$
(7) $\Gamma, \Gamma' \vdash s : \diamond$
(8) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma]$
(9) $\Gamma, \Gamma'; r \vdash \{F\} c : \mathtt{void}\{G\}$

We split cases according to the shape of c. Unless c is of the form $c = hc; c'$, the reduction rule is one of (Red Dcl), (Red Fin Dcl), or (Red Return). Because these cases are straightforward, we omit them here, and assume from this point on that c is of the form $c = hc; c'$:

(10) $c = hc; c'$
(11) $\Gamma, \Gamma'; r \vdash \{F\} hc \{F'\}$
(12) $\Gamma, \Gamma'; r \vdash \{F'\} c' : \mathtt{void}\{G\}$

Let $\mathcal{D}$ be the proof tree of $(\Gamma, \Gamma'; r \vdash \{F\} hc \{F'\})$. By Lemma 12, we may assume that each path to the root of $\mathcal{D}$ ends in a sequence of applications of (Consequence) and (Exists) preceded by exactly one application of (Frame). We induct on $\mathcal{D}$'s height:

Case 1, $\mathcal{D}$ ends in (Consequence): In this case, we have:

(1.1) $\Gamma, \Gamma'; r; F \vdash H$
(1.2) $\Gamma, \Gamma'; r \vdash \{H\} hc \{H'\}$
(1.3) $\Gamma, \Gamma'; r; H' \vdash F'$

From $(\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma])$ and $(\Gamma, \Gamma'; r; F \vdash H)$, it follows that:

(1.4) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma]$

From $(\Gamma, \Gamma'; r; H' \vdash F')$ and $(\Gamma, \Gamma'; r \vdash \{F'\} c' : \mathtt{void}\{G\})$, it follows that:

(1.5) $\Gamma, \Gamma'; r \vdash \{H'\} c' : \mathtt{void}\{G\}$

The height of $(\Gamma, \Gamma'; r \vdash \{H\} hc \{H'\})$'s proof tree is one less than $\mathcal{D}$'s height. In the proof tree of $st : \diamond$, we replace (12) by (1.4), (15) by (1.2), and (16) by (1.5). The resulting tree is a proof tree of $st : \diamond$. By induction hypothesis we obtain $st' : \diamond$.

Case 2, $\mathcal{D}$ ends in (Exists): In this case, we have:

(2.1) $F = (\mathtt{ex}\ T\ \alpha)(H)$
(2.2) $\Gamma, \Gamma', \alpha : T; r \vdash \{H\} hc \{H'\}$
(2.3) $F' = (\mathtt{ex}\ T\ \alpha)(H')$

From $(\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma])$, it follows that there is a π such that $(\Gamma[\sigma]_{\mathsf{hp}} \vdash \pi : T[\sigma])$ and $(\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma][\pi/\alpha])$. Let $\Gamma'' = (\Gamma', \alpha : T)$ and $\sigma' = (\sigma, \pi/\alpha)$. Then:

(2.4) $\Gamma \vdash \sigma' : \Gamma''$
(2.5) $\Gamma, \Gamma'' \vdash s : \diamond$

(2.6) $\Gamma[\sigma'] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma']$

From $(\Gamma, \Gamma''; r; H' \vdash F')$ and $(\Gamma, \Gamma'; r \vdash \{F'\}c' : \texttt{void}\{G\})$, it follows that:

(2.7) $\Gamma, \Gamma''; r \vdash \{H'\}c' : \texttt{void}\{G\}$

The height of $(\Gamma, \Gamma''; r \vdash \{H\}hc\{H'\})$'s proof tree is one less than $\mathcal{D}$'s height. In the proof tree of $st : \diamond$, we replace (7) by (2.4), (11) by (2.5), (12) by (2.6), (15) by (2.2), (16) by (2.7). The resulting tree is a proof tree of $st : \diamond$. By induction hypothesis, $st' : \diamond$.

Case 3, $\mathcal{D}$ ends in (Frame) preceded by a non-structural rule: We split this case into subcases according to the reduction rules. Because they are routine, we omit the cases (Red Op), (Red Return), (Red If True), (Red If False), and (Red No Op). Details can be found in the techreport accompanying our AMAST publication [54].

Case 3.1, (Red Get):

$$\frac{s' = s[\ell \mapsto h(p)_2(f)]}{\langle h,\ \ell = p.f;\ c', s\rangle \ \rightarrow\ \langle h,\ c', s'\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.1.1) $c = \ell = p.f;\ c'$
(3.1.2) $st' = \langle h,\ ts \mid c', s'\rangle$
(3.1.3) $s' = s[\ell \mapsto h(p)_2(f)]$

$\mathcal{D}$ ends in (Frame) preceded by (Get). From the premises of these rules, we obtain H, π and u such that the following statements hold:

(3.1.4) $F = \texttt{PointsTo}(p.f, \pi, u) * H$
(3.1.5) $\Gamma, \Gamma' \vdash p.f : (\Gamma)(\ell)$
(3.1.6) $\ell \notin H$
(3.1.7) $\Gamma, \Gamma'; r \vdash \{\texttt{PointsTo}(p.f, \pi, u) * H\}\ell = p.f\{\texttt{PointsTo}(p.f, \pi, u) * H * \ell == u\}$
(3.1.8) $\Gamma, \Gamma'; r \vdash \{\texttt{PointsTo}(p.f, \pi, u) * H * \ell == u\}c' : \texttt{void}\{G\}$

From (3.1.4), we deduce that $h(p)_2(f)$ is defined and that $h(p)_2(f) = u$. From (3.1.5), we obtain that $(\Gamma[\sigma] \vdash p.f : \Gamma(\ell)[\sigma])$. Because $(\Gamma[\sigma] \vdash h : \diamond)$, we then get $(\Gamma[\sigma] \vdash h(p)_2(f) : \Gamma(\ell)[\sigma])$. Then $(\Gamma, \Gamma' \vdash h(p)_2(f) : \Gamma(\ell))$, by Lemma 3. Thus, we have:

(3.1.9) $\Gamma, \Gamma' \vdash s' : \diamond$

From (12), we get $h(p)_2(f) = [\![u]\!]^h_s$. In addition, we have $[\![u]\!]^h_s = [\![u]\!]^h_{s'}$. On the other hand, we have $[\![\ell]\!]^h_{s'} = s'(\ell) = h(p)_2(f)$. Therefore, $[\![\ell]\!]^h_{s'} = h(p)_2(f) = [\![u]\!]^h_s = [\![u]\!]^h_{s'}$. It follows that (where $\mathbf{0}$ denotes the permission table that map all arguments to 0):

(3.1.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathbf{0}; s' \models \ell == u$

Because $\mathcal{R} * \mathbf{0} = \mathcal{R}$, we can combine (12) and (3.1.10) to obtain:

(3.1.11) $\Gamma, \Gamma'[\sigma] \vdash \mathcal{E}; \mathcal{R}; s' \models (\texttt{PointsTo}(p.f, \pi, u) * H)[\sigma] * \ \ell == u$

Because $(\ell == u)[\sigma] = \ell == u$ (recall that σ's domain is LogVar), we apply (State) to (3.1.11), (3.1.8), and (3.1.9). We obtain $st' : \diamond$.

Case 3.2, (Red Set):

$$\frac{h' = h[p.f \mapsto w]}{\langle h,\ p.f = w;\ c', s\rangle \ \rightarrow\ \langle h',\ c', s\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.2.1) $c = p.f = w;\ c'$
(3.2.2) $st' = \langle h',\ c', s\rangle$
(3.2.3) $h' = h[p.f \mapsto w]$

$\mathcal{D}$ ends in (Frame) preceded by (Fld Set). From the rule premises, we obtain F' and E such that:

(3.2.4) $F = \texttt{PointsTo}(p.f, 1, T) * H$
(3.2.5) $\Gamma, \Gamma' \vdash p : C\texttt{<}\bar{\pi}\texttt{>}$
(3.2.6) $T\ f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>})$
(3.2.7) $\Gamma, \Gamma' \vdash w : T$
(3.2.8) $\Gamma, \Gamma'; r \vdash \{\texttt{PointsTo}(p.f, 1, w) * H\}c' : \texttt{void}\{G\}$

From (12) and (3.2.4), we know there exist $\mathcal{R}', \mathcal{R}''$ such that $\mathcal{R} = \mathcal{R}' * \mathcal{R}''$ and:

(3.2.9) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models \texttt{PointsTo}(p.f, 1, T[\sigma])$
(3.2.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}''; s \models H[\sigma]$

By inverting $(\Gamma, \Gamma' \vdash p : C\texttt{<}\bar{\pi}\texttt{>})$ we obtain $(\Gamma, \Gamma')(p) <: C\texttt{<}\bar{\pi}\texttt{>}$. Because p is an object id and $\mathsf{dom}(\Gamma') \subseteq \mathsf{LogVar}$, we know that $p \notin \mathsf{dom}(\Gamma')$. Therefore, $\Gamma(p) <: C\texttt{<}\bar{\pi}\texttt{>}$. Let $h' = \mathcal{R}'_{\mathsf{hp}}$. Because $(\Gamma[\sigma] \vdash h' : \diamond)$, we know that $\Gamma[\sigma] = \mathsf{fst}(h')$. From $\Gamma[\sigma] = \mathsf{fst}(h')$ and $\Gamma(p) <: C\texttt{<}\bar{\pi}\texttt{>}$ it follows that $h'(p)_1 = (\Gamma[\sigma])(p) <: C\texttt{<}\bar{\pi}\texttt{>}[\sigma]$. From this and (3.2.6) it follows that $T[\sigma]\ f \in \mathsf{fld}(h'(p)_1)$. Applying substitutivity to (3.2.7), we get $(\Gamma[\sigma] \vdash w : T[\sigma])$. Furthermore, we have $\mathcal{R}'_{\mathsf{perm}}(p, f) = 1$ by (3.2.9).

From (3.2.9), it follows that:

(3.2.11) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'[p.f \mapsto w]; s \models \texttt{PointsTo}(p.f, 1, w)[\sigma]$

By Lemma 10, we know that $\mathcal{R}'[p.f \mapsto w] \# \mathcal{R}''$ and $\mathcal{R}'[p.f \mapsto w] * \mathcal{R}'' = \mathcal{R}[p.f \mapsto w]$. From (3.2.10) and (3.2.11), it then follows that:

(3.2.12) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}[p.f \mapsto w]; s \models (\texttt{PointsTo}(p.f, 1, w) * H)[\sigma]$

From (3.2.12), (3.2.8), and $\mathcal{R}[p.f \mapsto w]_{\mathsf{hp}} = h[p.f \mapsto w] = h'$, we apply (State) and obtain $\langle h', c', s\rangle : \diamond$.

Case 3.3, (Red New):

$$\frac{p \notin \mathsf{dom}(h)\quad h' = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))]\quad s' = s[\ell \mapsto p]\quad \boxed{l' = l[o \mapsto \mathsf{free}]}}{\langle h,\ \boxed{l}\ ,\ ts \mid \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c', s\rangle\ \rightarrow\ \langle h',\ \boxed{l'}\ ,\ ts \mid c', s'\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.3.1) $c = \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c'$
(3.3.2) $st' = \langle h',\ ts \mid c', s'\rangle$
(3.3.3) $h' = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))]$
(3.3.4) $s' = s[\ell \mapsto p]$

$\mathcal{D}$ ends in (Frame) preceded by (New). From the premises of these rules, we obtain:

(3.3.5) $F = \texttt{true} * H$

(3.3.6) $C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \in ct$
(3.3.7) $\Gamma, \Gamma' \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}]$
(3.3.8) $C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)$
(3.3.9) $\Gamma, \Gamma'; r \vdash \{\ell.\texttt{init} * C\ \texttt{classof}\ \ell * H\} c' : \texttt{void}\{G\}$
(3.3.10) $\ell \notin H$

By substitutivity, we get $(\Gamma[\sigma] \vdash \bar{\pi}[\sigma] : \bar{T}[\bar{\pi}/\bar{\alpha}][\sigma])$. Because $\mathsf{dom}(\Gamma') \cap \mathsf{fv}(\bar{\pi}) = \emptyset$ (assumption (8)) and $\mathsf{dom}(\Gamma') = \mathsf{dom}(\sigma)$, we get $(\Gamma[\sigma] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])$. Because $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$, it follows that $(\Gamma_{\mathsf{hp}}[\sigma] \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])$. Because $\Gamma_{\mathsf{hp}} \vdash \diamond$, we know that $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$, thus $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$, thus $(\Gamma_{\mathsf{hp}} \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\bar{\alpha}])$, thus:

(3.3.11) $\Gamma_{\mathsf{hp}} \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond$

Let $\Gamma_p = (\Gamma, p : C\texttt{<}\bar{\pi}\texttt{>})$ and $\mathcal{R}' = \mathsf{initrsc}(\Gamma[\sigma], p, C\texttt{<}\bar{\pi}\texttt{>})$. Thus, the premises for Lemma 11 are satisfied and we obtain:

(3.3.12) $\Gamma_p[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models p.\texttt{init}$

Furthermore, we have:

(3.3.13) $\Gamma_p[\sigma] \vdash \mathcal{E}; ((p \mapsto C\texttt{<}\bar{\pi}\texttt{>}, \emptyset), \mathbf{0}); s \models C\ \texttt{classof}\ p$

Because, by $\mathcal{R}'$'s and $*$'s definitions, we have $\mathcal{R}' * ((p \mapsto C\texttt{<}\bar{\pi}\texttt{>}, \emptyset), \mathbf{0}) = \mathcal{R}'$, it follows that:

(3.3.14) $\Gamma_p[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models p.\texttt{init} * C\ \texttt{classof}\ p$

By resource axiom (b) for $\mathcal{R}$, we have $\mathcal{R}_{\mathsf{perm}}(p, k) = 0$ for all $k \in \mathsf{FieldId}$. It follows that $\mathcal{R} \# \mathcal{R}'$. Thus:

(3.3.15) $\Gamma_p[\sigma] \vdash \mathcal{E}; \mathcal{R} * \mathcal{R}'; s \models H[\sigma] * p.\texttt{init} * C\ \texttt{classof}\ p$

Because, by (3.3.10), ℓ does not occur in $H[\sigma] * p.\texttt{init} * C\ \texttt{classof}\ p$, we can update s at ℓ:

(3.3.16) $\Gamma_p[\sigma] \vdash \mathcal{E}; \mathcal{R} * \mathcal{R}'; s' \models H[\sigma] * p.\texttt{init} * C\ \texttt{classof}\ p$

Because $s'(\ell) = p$, then:

(3.3.17) $\Gamma_p[\sigma] \vdash \mathcal{E}; \mathcal{R} * \mathcal{R}'; s' \models H[\sigma] * \ell.\texttt{init} * C\ \texttt{classof}\ \ell$

Because (3.3.11), (3.3.17), (3.3.9) and $(\mathcal{R} * \mathcal{R}')_{\mathsf{hp}} = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))] = h'$ hold, we can apply (State) to obtain $\langle h', c', s\rangle : \diamond$.

Case 3.4, (Red Call):

$$\frac{\begin{array}{c} m \notin \{\texttt{fork}, \texttt{join}, \texttt{wait}, \texttt{notify}\} \\ h(p)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\imath_0, \bar{\imath}).c_m \quad c'' = c_m[p/\imath_0, \bar{v}/\bar{\imath}] \end{array}}{\langle h,\ l\ ,\ ts \mid \ell = p.m(\bar{v});\ c', s\rangle \ \rightarrow\ \langle h,\ l\ ,\ ts \mid \ell \leftarrow c'';\ c', s\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.4.1) $c = \ell = p.m\texttt{<}\bar{\pi}\texttt{>}(\bar{v});\ c'$
(3.4.2) $st' = \langle h,\ ts \mid \ell \leftarrow c'';\ c', s\rangle$

$\mathcal{D}$ ends in (Frame) preceded by (Call). From the rule premises, we obtain:

(3.4.3) $F = E * J$
(3.4.4) $\mathsf{mtype}(m, t\texttt{<}\bar{\pi}''\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ E; \texttt{ensures}\ (\texttt{ex}\ U\ \alpha'')(H);\ U\ m(t\texttt{<}\bar{\pi}''\texttt{>}\ \imath_0; \bar{V}\ \bar{\imath})$
(3.4.5) $\sigma' = (p/\imath_0, \bar{\pi}/\bar{\alpha}, \bar{v}/\bar{\imath})$

(3.4.6) $\Gamma, \Gamma' \vdash p, \bar{\pi}, \bar{v} : t\texttt{<}\bar{\pi}''\texttt{>}, \bar{T}[\sigma'], \bar{V}[\sigma']$
(3.4.7) $U[\sigma'] <: \Gamma(\ell)$
(3.4.8) $\Gamma, \Gamma'; r \vdash \{J * \texttt{(ex}\, U[\sigma']\ \alpha'')\texttt{(}\alpha'' == \ell * H[\sigma']\texttt{)}\}c' : \texttt{void}\{G\}$
(3.4.9) $\ell \notin J$

In this case, (12) gives us[1]:

(3.4.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models E[\sigma'; \sigma] * J[\sigma]$

Because of method inheritance and subtyping, we do not know in which class m is implemented. We do a case split on the two possible cases: (1) m is implemented in a class B such that $t \preceq B$ or (2) m is implemented in a class D such that $C \preceq D \preceq t$ (recall that, by $h(p)_1 = C\texttt{<}\bar{\pi}'\texttt{>}$, C is p's dynamic class).

Case 3.4.1, m is implemented in a class B such that $t \preceq B$: Without losing generality, suppose that $t\texttt{<}\bar{U}\ \bar{\alpha}'\texttt{>} \in ct$ and $B\texttt{<}\bar{W}\ \bar{\alpha}''\texttt{>} \in ct$. In this case, we have:

(3.4.1.1) $\mathsf{mbody}(m, t\texttt{<}\bar{\pi}''\texttt{>}) = \mathsf{mbody}(m, B\texttt{<}\bar{\alpha}''[\bar{\pi}''/\bar{\alpha}']\texttt{>})$
(3.4.1.2) $\mathsf{mtype}(m, t\texttt{<}\bar{\pi}''\texttt{>}) = \mathsf{mtype}(m, B\texttt{<}\bar{\alpha}''[\bar{\pi}''/\bar{\alpha}']\texttt{>})$

We have assumed that $ct : \diamond$. By (3.4.4), (3.4.1.1) and (3.4.1.2), it follows that we can take the premise of rule (Mth) for m in B to obtain:

(3.4.1.3) $\bar{\alpha} : \bar{T}, \bar{\alpha}'' : \bar{W}, \bar{\imath} : \bar{V}, \imath_0 : B\texttt{<}\bar{W}\texttt{>}; \imath_0 \vdash \begin{array}{c}\{E[\sigma'] * \imath_0 \neq \texttt{null}\} \\ c'' : U[\sigma'] \\ \{\texttt{(ex}\, U[\sigma']\ \alpha'')\texttt{(}H[\sigma']\texttt{)}\}\end{array}$

Let $\Gamma'' = \bar{\pi} : \bar{T}[\sigma'], \bar{\pi}'' : \bar{W}[\sigma'], \bar{v} : \bar{V}[\sigma'], p : B\texttt{<}\bar{W}[\sigma']\texttt{>}$. By substituting actual class parameters for formal class parameters and actual method parameters for formal method parameters, we obtain:

(3.4.1.4) $\Gamma''; p \vdash \{E[\sigma'] * p \neq \texttt{null}\}c'' : U[\sigma']\{\texttt{(ex}\, U[\sigma']\ \alpha'')\texttt{(}H[\sigma']\texttt{)}\}$

By weakening the type environment from Γ'' to (Γ, Γ') (by (3.4.6)) and the frame rule, we obtain:

(3.4.1.5) $\Gamma, \Gamma'; p \vdash \begin{array}{c}\{J * E[\sigma'] * p \neq \texttt{null}\} \\ c'' : U[\sigma'] \\ \{\texttt{(ex}\, U[\sigma']\ \alpha'')\texttt{(}J * H[\sigma']\texttt{)}\}\end{array}$

Now we can apply the derived rule for "bind" (Lemma 7) to (3.4.1.5) and (3.4.8):

(3.4.1.6) $\Gamma, \Gamma'; p \vdash \{J * E'[\sigma'] * p \neq \texttt{null}\}\ell \leftarrow c''; c' : \texttt{void}\{G\}$

Because (3.4.2.8), (3.4.2.7) and (3.4.1.6) hold, we can apply (State) to obtain:

(3.4.1.7) $\langle h, \ell \leftarrow c''; c', s\rangle : \diamond$

Case 3.4.2, m is implemented in a class D such that $C \preceq D \preceq t$: Without losing generality, suppose $C\texttt{<}\bar{U}\ \bar{\alpha}'\texttt{>} \in ct$ and $D\texttt{<}\bar{W}\ \bar{\alpha}''\texttt{>} \in ct$. Because $D \preceq t$, we know there exists $\bar{\pi}'''$ such that $D\texttt{<}\bar{\pi}'''\texttt{>} <: t\texttt{<}\bar{\pi}''\texttt{>}$. Therefore, by monotonicity of mtype, $\mathsf{mtype}(m, D\texttt{<}\bar{\pi}'''\texttt{>}) <: \mathsf{mtype}(m, t\texttt{<}\bar{\pi}''\texttt{>})$. Then, by definition of method subtyping, we get:

(3.4.2.1) $\mathsf{mtype}(m, D\texttt{<}\bar{\pi}'''\texttt{>}) =$
$\texttt{<}\bar{T}'\ \bar{\alpha}, \bar{W}\ \bar{\alpha}'\texttt{>}\ \texttt{requires}\ E'; \texttt{ensures (ex}\, U'\ \alpha'')\texttt{(}H'\texttt{)};\ U'\ m\texttt{(}C\texttt{<}\bar{\pi}'\texttt{>}\ \imath_0; \bar{V}'\ \bar{\imath}\texttt{)}$

[1] We use the semicolon for substitution composition: $(\sigma'; \sigma)(x) \triangleq \sigma'(x)[\sigma]$

(3.4.2.2) $\bar{T} <: \bar{T}'$, $U' <: U$, $\bar{V} <: \bar{V}'$

(3.4.2.3) $\Gamma_{\mathsf{hp}}, \imath_0 : D\texttt{<}\bar{\pi}'''\texttt{>}; \imath_0; \texttt{true} \vdash$
$\texttt{(fa}\ \bar{T}\ \bar{\alpha}\texttt{)(fa}\ \bar{V}\ \bar{\imath}\texttt{)(}E\ \texttt{-*}\ \texttt{(ex}\ \bar{W}\ \bar{\alpha}'\texttt{)(}E'\ \texttt{*}\ \texttt{(fa}\ U'\ \alpha''\texttt{)(}H'\ \texttt{-*}\ H\texttt{)))}$

To abbreviate, let $H'' = \texttt{(fa}\ U'\ \alpha''\texttt{)(}H'\ \texttt{-*}\ H\texttt{)}$. Applying substitutivity and (Fa Elim) to (3.4.2.3), we obtain:

(3.4.2.4) $\Gamma_{\mathsf{hp}}; p; \texttt{true} \vdash E[\sigma'; \sigma]\ \texttt{-*}\ \texttt{(ex}\ \bar{W}[\sigma'; \sigma]\ \bar{\alpha}'\texttt{)(}E'[\sigma'; \sigma]\ \texttt{*}\ H''[\sigma'; \sigma]\texttt{)}$

From (3.4.10) and (3.4.2.4), it follows that there exist $\bar{\pi}''''$ and σ'' such that:

(3.4.2.5) $\sigma'' = (\sigma, \bar{\alpha}' \mapsto \bar{\pi}''')$

(3.4.2.6) $\Gamma[\sigma''] \vdash \bar{\pi}'''' : \bar{W}[\sigma'; \sigma'']$

(3.4.2.7) $\Gamma[\sigma''] \vdash \mathcal{E}; \mathcal{R}; s \models J[\sigma''] * E'[\sigma'; \sigma''] * H''[\sigma'; \sigma'']$

Let $\Gamma'' = (\Gamma', \bar{\alpha} : \bar{W}[\sigma'])$. By (3.4.2.6), we get:

(3.4.2.8) $\Gamma \vdash \sigma'' : \Gamma''$

Because $C \preceq D$, by mbody's definition:

(3.4.2.9) $\mathsf{mbody}(m, C\texttt{<}\bar{\pi}'\texttt{>}) = \mathsf{mbody}(m, D\texttt{<}\bar{\alpha}''[\bar{\pi}'/\bar{\alpha}']\texttt{>})$

We have assumed that $ct : \diamond$. Because (3.4.2.9) holds, by taking the premise of rule (Mth) for m in D and substitute actual class parameters for formal class parameters and actual method parameters for formal method parameters, we obtain:

(3.4.2.10) $\Gamma, \Gamma''; p \vdash \{E'[\sigma'] * p \neq \texttt{null}\} c'' : U'[\sigma']\{\texttt{(ex}\ U'[\sigma']\ \alpha''\texttt{)(}H'[\sigma']\texttt{)}\}$

From (3.4.2.10) and (3.4.2.9) By the frame rule, we obtain:

(3.4.2.11) $\Gamma, \Gamma''; p \vdash \{J * H''[\sigma'] * E'[\sigma'] * p \neq \texttt{null}\}$
$c'' : U'[\sigma']$
$\{\texttt{(ex}\ U'[\sigma']\ \alpha''\texttt{)(}J * H''[\sigma'] * H'[\sigma']\texttt{)}\}$

We weaken (3.4.8) by extending the type environment to (Γ, Γ''):

(3.4.2.12) $\Gamma, \Gamma''; r \vdash \{J\ \texttt{*}\ \texttt{(ex}\ U[\sigma']\ \alpha''\texttt{)(}\alpha''\ \texttt{==}\ \ell\ \texttt{*}\ H[\sigma']\texttt{)}\} c' : \texttt{void}\{G\}$

Using the natural deduction rules, one can show the following;

(3.4.2.13) $\Gamma, \Gamma''; r; \texttt{(ex}\ U'[\sigma']\ \alpha''\texttt{)(}J * H''[\sigma'] * H'[\sigma']\texttt{)}$
$\vdash J\ \texttt{*}\ \texttt{(ex}\ U[\sigma']\ \alpha''\texttt{)(}\alpha''\ \texttt{==}\ \ell\ \texttt{*}\ H[\sigma']\texttt{)}$

Thus, by logical consequence (Lemma 6), we get:

(3.4.2.14) $\Gamma, \Gamma''; r \vdash \{\texttt{(ex}\ U'[\sigma']\ \alpha''\texttt{)(}J * H''[\sigma'] * H'[\sigma']\texttt{)}\} c' : \texttt{void}\{G\}$

Now we can apply the derived rule for "bind" (Lemma 7) to (3.4.2.11) and (3.4.2.14):

(3.4.2.15) $\Gamma, \Gamma''; p \vdash \{J * H''[\sigma'] * E'[\sigma'] * p \neq \texttt{null}\} \ell \leftarrow c''; c' : \texttt{void}\{G\}$

Because (3.4.2.8), (3.4.2.7) and (3.4.2.15) hold, we can apply (State) to obtain:

(3.4.2.16) $\langle h, \ell \leftarrow c''; c', s\rangle : \diamond$

This concludes the proof. □

6.1.3 Null Error Freeness and Partial Correctness

We can now easily prove several corollaries of the preservation theorem.

Lemma 13. *If* $(ct, c) : \diamond$, *then* $\mathsf{init}(c) : \diamond$.

Proof. Suppose $(ct, c) : \diamond$. By definition, this means $ct : \diamond$ and $\emptyset; \texttt{null} \vdash \{\texttt{true}\}c : \texttt{void}\{\texttt{true}\}$. Let $\mathcal{R} = (\emptyset, \mathbf{0})$. Let $\mathcal{E}$ be some predicate environment such that $\mathcal{F}_{ct}(\mathcal{E}) = \mathcal{E}$ (which exists by Theorem 1). Then $(\emptyset \vdash \mathcal{E}; \mathcal{R}; \emptyset \models \texttt{true})$. Now it is easy to check that the premises of (State) are satisfied (pick $\sigma = \Gamma = \Gamma' = s = \emptyset$, $r = \texttt{null}$, and $F = G = \texttt{true}$). Thus, $\mathsf{init}(c) : \diamond$, by (State). □

Recall that a head command hc is called a *null error* iff $hc = (\ell\texttt{=null}.f)$ or $hc = (\mathit{fin}\ \texttt{null}.f\texttt{=}v)$ or $hc = (\ell\texttt{=null}.m\texttt{<}\bar{\pi}\texttt{>(}\bar{v}\texttt{)})$ for some $\ell, \mathit{fin}, f, v, m, \bar{\pi}, \bar{v}$.

Proof of Theorem 4 (Verified Programs are Null Error Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, hc; c, s\rangle$, *then* hc *is not a null error.*

Proof. Let $(ct, c) : \diamond$, $st = \langle h, hc; c, s\rangle$, and $\mathsf{init}(c) \rightarrow^*_{ct} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 13) and preservation (Theorem 3), we know that $st : \diamond$. Suppose, towards a contradiction, that hc is a null error. Then $hc = (\ell\texttt{=null}.f)$ or $hc = (\texttt{null}.f\texttt{=}v)$ or $hc = (\ell\texttt{=null}.m\texttt{<}\bar{\pi}\texttt{>(}\bar{v}\texttt{)})$.

Suppose first that $hc = (\ell\texttt{=null}.f)$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathcal{E}$, $\mathcal{R}$, s, π, u such that either $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models \texttt{PointsTo(null}.f, \pi, u\texttt{)}$. But by definition of $\models$, this statement cannot hold.

Suppose now that $hc = (\texttt{null}.f\texttt{=}v)$ An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathcal{E}$, $\mathcal{R}$, s, T such that $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models \texttt{PointsTo(null}.f, 1, T\texttt{)}$. But this is false, by definition of $\models$.

Suppose finally that $hc = (\ell\texttt{=null}.m\texttt{<}\bar{\pi}\texttt{>(}\bar{v}\texttt{)})$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathcal{E}$, $\mathcal{R}$, s such that $\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models \texttt{null != null}$, which is obviously false. □

Proof of Theorem 5 (Partial Correctness).
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, \texttt{assert(}F\texttt{)}; c, s\rangle$, *then* $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$ *for some* $\Gamma, \mathcal{E} = \mathcal{F}_{ct}(\mathcal{E}), \mathcal{P}$ *and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

Proof. Let $(ct, c) : \diamond$, $st = \langle h, \texttt{assert(}F\texttt{)}; c, s\rangle$, and $\mathsf{init}(c) \rightarrow^*_{ct} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 13) and preservation (Theorem 3), we know that $st : \diamond$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathcal{E} = \mathcal{F}_{ct}(\mathcal{E})$, $\mathcal{R}$, $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$ such that $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}); s \models F[\sigma])$. □

6.2 Soundness of Chapter 4's Verification System

First, we state some lemmas that are needed in the next section.

6.2.1 Properties

Symbolic Binary Fractions. We define functions $\mathsf{perm2bits} : \mathsf{SpecVal} \rightharpoonup \mathsf{Bits}$ and $\mathsf{perm2binfr} : \mathsf{SpecVal} \rightharpoonup \mathsf{BinFrac}$ to map fractional permissions to binary fractions:

$$\mathsf{perm2bits}(\texttt{split}(\pi)) \triangleq \begin{cases} 1 & \text{iff } \pi = 1 \\ 0, \mathsf{perm2bits}(\pi) & \text{otherwise} \end{cases}$$

$$\mathsf{perm2binfr}(\pi) \triangleq \begin{cases} \mathsf{all} & \text{iff } \pi = 1 \\ \mathsf{fr}(\mathsf{perm2bits}(\pi)) & \text{otherwise} \end{cases}$$

For example, we have $\mathsf{perm2binfr}(\texttt{split}(\texttt{split}(1))) = \mathsf{fr}(0,1)$ and $[\![\mathsf{fr}(0,1)]\!] = 1/4$. Functions perm2bits and perm2binfr are partial because they only make sense on fractional permissions (recall that the set SpecVal also contains Java values).

Lemma 14 (perm2binfr Is an Equivalence Relation)**.** *If* $\mathsf{perm2binfr}(\pi)$ *is defined, then* $[\![\pi]\!] = [\![\mathsf{perm2binfr}(\pi)]\!]$.

Proof. By induction on π. □

Lemma 15 (Subtraction of Symbolic Fractions)**.** *If* $[\![fr_2]\!] \leq [\![fr_1]\!]$*, then there exists a unique symbolic fraction* $fr_1 - fr_2$ *such that* $(fr_1 - fr_2) + fr_2 = fr_1$.

Proof. If $[\![fr_1]\!] - [\![fr_2]\!] = 0$, we define $fr_1 - fr_2 \triangleq \mathsf{fr}()$. If $[\![fr_1]\!] - [\![fr_2]\!] = 1$, we define $fr_1 - fr_2 \triangleq \mathsf{all}$. Otherwise, we represent $[\![fr_1]\!] - [\![fr_2]\!]$ as $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$ where $bit_n = 1$, and we define $fr_1 - fr_2 \triangleq \mathsf{fr}(bit_1, \ldots, bit_n)$. By construction, we have $[\![fr_1 - fr_2]\!] + [\![fr_2]\!] = [\![fr_2]\!]$. Then $(fr_1 - fr_2) + fr_2 = fr_2$ because, by our definition of addition, $(fr_1 - fr_2) + fr_2$ is the only symbolic fraction fr such that $[\![fr_1 - fr_2]\!] + [\![fr_2]\!] = [\![fr]\!]$. □

Lemma 16 (Distributivity of Scalar Multiplication)**.** *If* $fr_1 + fr_2$ *exists and* $\Gamma; v; \texttt{true} \vdash o.P\texttt{<}\pi\texttt{>} \mathrel{\texttt{*-*}} o.P\texttt{<split(}\pi\texttt{)>} * o.P\texttt{<split(}\pi\texttt{)>}$ *holds, then:*

$$\Gamma; v; \texttt{true} \vdash (fr_1 + fr_2) \cdot o.P\texttt{<1>} \mathrel{\texttt{*-*}} fr_1 \cdot o.P\texttt{<1>} * \ fr_2 \cdot o.P\texttt{<1>}$$

Proof. Similar to the proof of Lemma 92 in [54, §T]. For this proof to apply here, we first remark that $\Gamma; v; \texttt{true} \vdash o.P\texttt{<}\pi\texttt{>} \mathrel{\texttt{*-*}} (o.P\texttt{<split(}\pi\texttt{)>} * o.P\texttt{<split(}\pi\texttt{)>})$ implies *supportedness* (as defined in [54]) of P. □

Semantics. The following lemma is needed to update the global join table after calling `join` (recall that the global join table is an upper bound on how much of threads's postconditions can be taken back).

Lemma 17 (Thread Joining)**.** *Let* $o \in \mathsf{dom}(h)$, $\mathcal{P}(o, \texttt{join}) \leq x \leq \mathcal{J}(o)$ *and* $\mathcal{J}' = \mathcal{J}[(o) \mapsto x]$.

If $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}); s \models F)$*, then* $(\Gamma \vdash \mathcal{E}; (h, \mathcal{P}, \mathcal{J}'); s \models F)$.

Proof. By induction on the structure of F. For cases $F = o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$ and $F = o.P\texttt{<}\bar{\pi}\texttt{>}$, we use axiom (b) for predicate environments. □

6.2.2 Preservation

Now, we update Theorem 3's proof to cope with the new cases of the operational semantics. We do not show how existing cases are updated to account for multiple threads: the only change is that there is an extra level of indirection between the top level resource and the resource for each thread (because states now include a thread pool instead of a single thread). Details can be found in [54].

Proof of Theorem 3 (Preservation). *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$*, then* $(st' : \diamond)$.

Proof.

(1) $ct : \diamond$ assumption
(2) $st : \diamond$ assumption
(3) $st \rightarrow st'$ assumption

An inspection of the reduction rules shows that st is of the following form.

(4) $st = \langle h,\ ts \mid o \text{ is } (s \text{ in } c)\rangle$

The proof of $(st : \diamond)$ ends in an application of (State), preceded by an application of (Cons Pool), preceded by an application of (Thread) for thread o. The rule (Cons Pool) includes the following premise:

(5) $\mathcal{R}_{ts} \vdash ts : \diamond$

The rule (Thread) has the following premises:

(6) $\mathcal{R}_{\mathsf{join}}(o) \leq [\![fr]\!]$
(7) $\Gamma \vdash \sigma : \Gamma'$
(8) $\mathsf{dom}(\Gamma') \cap \mathsf{cfv}(c) = \emptyset$
(9) $\Gamma, \Gamma' \vdash s : \diamond$
(10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma]$
(11) $\Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{fr \cdot o.\texttt{postJoin<1>}\}$

The proof structure is exactly as in Section 6.1.2. The only difference is that we have to complete case 3 with the new reduction cases (Red Fork) and (Red Join). In these cases, we can assume that the following statements hold:

(12) $c = hc;\ c'$
(13) $\Gamma, \Gamma'; r \vdash \{F\}hc\{F'\}$
(14) $\Gamma, \Gamma'; r \vdash \{F'\}c' : \texttt{void}\{fr \cdot o.\texttt{postJoin<1>}\}$

We now add cases (Red Fork) and (Red Join) to subcase 3 of Section 6.1.2's proof.

Case 3.5, (Red Fork):

$$\frac{h(p)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad p \notin (\mathsf{dom}(ts) \cup \{o\}) \quad \mathsf{mbody}(\texttt{run}, C\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{this}).c_r \quad c'' = c_r[p/\texttt{this}]}{\langle h,\ ts \mid o \text{ is } (s \text{ in } \ell = p.\texttt{fork()};\ c')\rangle \ \rightarrow\ \langle h,\ ts \mid o \text{ is } (s \text{ in } \ell = \texttt{null};\ c') \mid p \text{ is } (\emptyset \text{ in } c'')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.5.1) $c = \ell = p.\texttt{fork()};\ c'$
(3.5.2) $st' = \langle h,\ ts \mid o \text{ is } (s \text{ in } \ell = \texttt{null};\ c') \mid p \text{ is } (\emptyset \text{ in } c'')\rangle$

The last rules in (13)'s derivation are (Frame) preceded by (Call). From the rule premises, we obtain:

(3.5.3) $F = \texttt{this.preFork} * H.$
(3.5.4) $\Gamma, \Gamma' \vdash p : t\texttt{<}\bar{\pi}'\texttt{>}$
(3.5.5) $\texttt{void} <: \Gamma(\ell)$
(3.5.6) $\Gamma, \Gamma'; r \vdash \{H * (\texttt{ex void } \alpha')(\alpha' == \ell)\}c' : \texttt{void}\{fr \cdot o.\texttt{postJoin<1>}\}$
(3.5.7) $\ell \notin H$

Because of method inheritance and subtyping, we do not know in which class `run` is implemented. Two cases are possible: (1) `run` is implemented in a class B such that $t \preceq B$ or (2) `run` is implemented in a class D such that $C \preceq D \preceq t$ (recall that, by $h(p)_1 = C\texttt{<}\bar{\pi}\texttt{>}$, C is p's dynamic class). From now on, we assume case (2) holds and omit case (1) (it is simpler than case (2)).

Without loss of generality, let $C\texttt{<}\bar{W}\ \bar{\alpha}\texttt{>} \in ct$, $D\texttt{<}\bar{T}\ \bar{\alpha}'\texttt{>} \in ct$, and $t\texttt{<}\bar{V}\ \bar{\alpha}''\texttt{>} \in ct$. By mbody's definition, we know that:

(3.5.8) $\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = \mathsf{mbody}(m, D\texttt{<}\bar{\alpha}'[\bar{\pi}/\bar{\alpha}]\texttt{>})$

Because (3.5.8) holds, from the premises of rule (Mth) for $D.\texttt{run}$, we obtain (recall that logical parameters are forbidden for `run`, hence the simple statement):

$$(3.5.9)\quad \bar{\alpha}' : \bar{T}, \texttt{this} : D\texttt{<}\bar{\alpha}'\texttt{>}\,; \texttt{this} \vdash \begin{array}{c}\{\texttt{this.preFork} * \texttt{this != null}\} \\ c_r : \texttt{void} \\ \{\texttt{(ex void } \alpha'\texttt{)(this.postJoin<1>)}\}\end{array}$$

Because $D \preceq t$, we know that there exists $\bar{\pi}''$ such that $D\texttt{<}\bar{\pi}''\texttt{>} <: t\texttt{<}\bar{\pi}'[\bar{\pi}''/\bar{\alpha}']\texttt{>}$. Because $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, we know that $(\Gamma_{\mathsf{hp}} \vdash \bar{\pi}'[\bar{\pi}''/\bar{\alpha}'] : \bar{V})$. From the last two judgments, by subtyping and substitutivity, we obtain:

$$(3.5.10)\quad \bar{\alpha}' : \bar{T}[\bar{\pi}''/\bar{\alpha}'], \texttt{this} : t\texttt{<}\bar{\pi}'[\bar{\pi}''/\bar{\alpha}']\texttt{>}\,; \texttt{this} \vdash \begin{array}{c}\{\texttt{this.preFork} * \texttt{this != null}\} \\ c_r : \texttt{void} \\ \{\texttt{(ex void } \alpha'\texttt{)(this.postJoin<1>)}\}\end{array}$$

By applying substitutivity again (Lemma 5) to (3.5.10), we obtain:

(3.5.11) $\Gamma_{\mathsf{hp}}; p \vdash \{p.\texttt{preFork} * p\ \texttt{!= null}\} c'' : \texttt{void}\{\texttt{(ex void } \alpha'\texttt{)(}p.\texttt{postJoin<1>)}\}$

By the definition of scalar definition, it follows that:

(3.5.12) $\Gamma_{\mathsf{hp}}; p \vdash \{p.\texttt{preFork} * p\ \texttt{!= null}\} c'' : \texttt{void}\{\texttt{(ex void } \alpha'\texttt{)(all} \cdot p.\texttt{postJoin<1>)}\}$

From (12) and (3.5.3), we know there exist $\mathcal{R}_p$ and $\mathcal{R}_o$ such that:

(3.5.13) $\mathcal{R}_p * \mathcal{R}_o = \mathcal{R}$
(3.5.14) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}_p; s \models p.\texttt{preFork} * p\ \texttt{!= null}$
(3.5.15) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}_o; s \models H$

From (3.5.14) and (3.5.12), applying (Thread) yields:

(3.5.16) $\mathcal{R}_p \vdash p\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c'') : \diamond$

Applying admissibility of logical consequence (Lemma 6) to (3.5.6), we get:

(3.5.17) $\Gamma, \Gamma'; r \vdash \{H * \texttt{(ex void } \alpha'\texttt{)(}\alpha'\texttt{ == }\ell\texttt{)} * \ell\ \texttt{== null}\} c' : \texttt{void}\{fr \cdot o.\texttt{postJoin<1>}\}$

Then, by (3.5.7), we can apply (Var Set) to obtain:

(3.5.18) $\Gamma, \Gamma'; r \vdash \{H * \texttt{(ex void } \alpha'\texttt{)(}\alpha'\texttt{ == }\ell\texttt{)}\} \ell\texttt{ = null; } c' : \texttt{void}\{fr \cdot o.\texttt{postJoin<1>}\}$

The existential formula can be validated by instantiating α' by $s(\ell)$. Therefore, (3.5.15) gives us:

(3.5.19) $\Gamma, \Gamma' \vdash \mathcal{E}; \mathcal{R}_o; s \models H * \texttt{(ex void } \alpha'\texttt{)(}\alpha'\texttt{ == }\ell\texttt{)}$

From (3.5.19) and (3.5.18), by (Thread), it follows that:

(3.5.20) $\mathcal{R}_o \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{ = null; } c') : \diamond$

The rest is routine.

Case 3.6, (Red Join):

$\langle h,\ ts' \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } \ell = p.\texttt{join}();\ c') \mid p \mathsf{\ is\ } (s' \mathsf{\ in\ } v)\rangle \rightarrow \langle h,\ ts' \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } \ell = \texttt{null};\ c') \mid p \mathsf{\ is\ } (s' \mathsf{\ in\ } v)\rangle$

In this case, we can further instantiate c and st' as follows:

(3.6.1) $ts = ts' \mid p \mathsf{\ is\ } (s' \mathsf{\ in\ } v)$
(3.6.2) $c = \ell = p.\texttt{join}();\ c'$
(3.6.3) $st' = \langle h,\ ts' \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } \ell = \texttt{null};\ c') \mid p \mathsf{\ is\ } (s' \mathsf{\ in\ } v)\rangle$

The last rules in (13)'s derivation are (Frame) preceded by (Mth) (with $m = \texttt{join}$). From the rule premises we obtain:

(3.6.4) $F = \texttt{Join}(p, \pi') * H$
(3.6.5) $\Gamma, \Gamma' \vdash p : C'\texttt{<}\bar{\pi}'\texttt{>}$
(3.6.6) $\sigma' = (p/\texttt{this})$
(3.6.7) $\texttt{void} <: \Gamma(\ell)$
(3.6.8) $\Gamma, \Gamma'; r \vdash \begin{array}{c}\{H * (\texttt{ex void } \alpha')(\bar{\alpha}' == \ell * p.\texttt{postJoin<}\pi'\texttt{>})\} \\ c' : \texttt{void} \\ \{fr \cdot o.\texttt{postJoin<1>}\}\end{array}$
(3.6.9) $\ell \notin H$

From (12) and (3.6.4), we obtain:

(3.6.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Join}(p, \pi')$

By the semantics of $*$, there exist $\mathcal{R}^{o,1}$ and $\mathcal{R}^{o,2}$ such that:

(3.6.11) $\mathcal{R} = \mathcal{R}^{o,1} * \mathcal{R}^{o,2}$
(3.6.12) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^{o,1}; s \models H[\sigma]$
(3.6.13) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^{o,2}; s \models \texttt{Join}(p, \pi')$

The last of these statements means that:

(3.6.14) $[\![\pi']\!] \leq \mathcal{R}^{o,2}_{\mathsf{perm}}(p)$

Recall that $(\mathcal{R}_{ts} \vdash ts : \diamond)$, by (5), and $ts = ts' \mid p \mathsf{\ is\ } (s' \mathsf{\ in\ } v)$, by (3.6.1). The premises of the last rule of $(\mathcal{R}_{ts} \vdash ts : \diamond)$'s derivation are:

(3.6.15) $\mathcal{R}^{ts} = \mathcal{R}^{ts'} * \mathcal{R}^{p}$
(3.6.16) $\mathcal{R}^{ts'} \vdash ts' : \diamond$
(3.6.17) $\mathcal{R}^{p} \vdash p \mathsf{\ is\ } (s' \mathsf{\ in\ } v) : \diamond$

Let $\mathcal{J} = \mathcal{R}^{p}_{\mathsf{join}}$. Recall that the $*$-composition of two resources is only defined if they both have the same global join table. So $\mathcal{J} = \mathcal{R}^{ts}_{\mathsf{join}} = \mathcal{R}_{\mathsf{join}} = \mathcal{R}^{o}_{\mathsf{join}}$ holds, too. From $(\mathcal{R}^{p} \vdash p \mathsf{\ is\ } (s' \mathsf{\ in\ } v) : \diamond)$, by inverting (Thread), we know there exists fr_p such that these judgments hold:

(3.6.18) $\Gamma''[\sigma'''] \vdash \mathcal{E}; \mathcal{R}^{p}; s' \models fr_p \cdot p.\texttt{postJoin<1>}$
(3.6.19) $\mathcal{J}(p) \leq [\![fr_p]\!]$

We know that $\Gamma''_{\mathsf{hp}}[\sigma'''] = \Gamma''_{\mathsf{hp}} = h(o)_1 = \Gamma_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}[\sigma]$. Because $fr_p \cdot p.\texttt{postJoin<1>}$ does not contain free variables, we can restrict the stack in (3.6.18):

(3.6.20) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^{p}; \emptyset \models fr_p \cdot p.\texttt{postJoin<1>}$

We define: $\sigma'' = (\sigma', v/\alpha')$. Let $fr' \triangleq \mathsf{perm2binfr}(\pi')$. By axiom (b) for resources, we have that $\big(\lambda q.\mathcal{R}^{o,2}_{\mathsf{perm}}(q, \texttt{join})\big)(p) \leq \mathcal{J}(p)$ i.e., $\mathcal{R}^{o,2}_{\mathsf{perm}}(p, \texttt{join}) \leq \mathcal{J}(p)$. Combining all inequalities together, we obtain:

(3.6.21) $[\![\pi']\!] \leq \mathcal{R}^{o,2}_{\mathsf{perm}}(p, \texttt{join}) \leq \mathcal{J}(p) \leq [\![fr_p]\!]$

By Lemma 14 and Lemma 15, it follows that $fr_p - fr'$ exists. By standard arithmetic, we have $fr_p \cdot p.\texttt{postJoin<1>} = ((fr_p - fr') + fr') \cdot p.\texttt{postJoin<1>}$. Because `postJoin` is a group, we can apply distributivity (Lemma 16) and obtain:

(3.6.22) $((fr_p - fr') + fr') \cdot p.\texttt{postJoin<1>} \mathbin{*\!\!-\!\!*} (fr_p - fr') \cdot p.\texttt{postJoin<1>} * fr' \cdot p.\texttt{postJoin<1>}$

Therefore, (3.6.20) implies the following statement:

(3.6.23) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^p; \emptyset \models (fr_p - fr') \cdot p.\texttt{postJoin<1>} * fr' \cdot p.\texttt{postJoin<1>}$.

By definition of semantic validity, there exist $\mathcal{R}^{p,1}$ and $\mathcal{R}^{p,2}$ such that:

(3.6.24) $\mathcal{R}^p = \mathcal{R}^{p,1} * \mathcal{R}^{p,2}$
(3.6.25) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^{p,1}; \emptyset \models (fr_p - fr') \cdot p.\texttt{postJoin<1>}$
(3.6.26) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^{p,2}; \emptyset \models fr' \cdot p.\texttt{postJoin<1>}$

We now define:

(3.6.27) $\mathcal{J}' \triangleq \mathcal{J}[p \mapsto (\mathcal{J}(p) - [\![fr']\!])]$
(3.6.28) $(\mathcal{R}^{ts'})' \triangleq (\mathcal{R}^{ts'}_{\mathsf{hp}}, \mathcal{R}^{ts'}_{\mathsf{perm}}, \mathcal{J}')$
(3.6.29) $(\mathcal{R}^{o,1})' \triangleq (\mathcal{R}^{o,1}_{\mathsf{hp}}, \mathcal{R}^{o,1}_{\mathsf{perm}}, \mathcal{J}')$
(3.6.30) $(\mathcal{R}^{p,2})' \triangleq (\mathcal{R}^{p,2}_{\mathsf{hp}}, \mathcal{R}^{p,2}_{\mathsf{perm}}, \mathcal{J}')$
(3.6.31) $(\mathcal{R}^{p})' \triangleq (\mathcal{R}^{p,1}_{\mathsf{hp}}, \mathcal{R}^{p,1}_{\mathsf{perm}}, \mathcal{J}')$
(3.6.32) $(\mathcal{R}^{o})' \triangleq (\mathcal{R}^{o,1})' * (\mathcal{R}^{p,2})'$

It now suffices to show the following claims:

(3.6.33) $(\mathcal{R}^{ts'})' \vdash ts' : \diamond$ goal
(3.6.34) $(\mathcal{R}^{p})' \vdash p \;\mathsf{is}\; (s' \;\mathsf{in}\; v) : \diamond$ goal
(3.6.35) $(\mathcal{R}^{o})' \vdash o \;\mathsf{is}\; (s \;\mathsf{in}\; \ell = \texttt{null}; c') : \diamond$ goal

Goal (3.6.33) is a consequence of $(\mathcal{R}^{ts'} \vdash ts' : \diamond)$ and Lemma 17. To show goal (3.6.34) we use (3.6.25) and Lemma 17. So we are left with goal (3.6.35): By combining (3.6.12) and (3.6.26), we obtain:

(3.6.36) $\Gamma[\sigma] \vdash \mathcal{E}; (\mathcal{R}^{o,1})' * (\mathcal{R}^{p,2})'; \emptyset \models H[\sigma] * fr' \cdot p.\texttt{postJoin<1>}$

Because of (3.6.35), $\sigma'' = (\sigma', v/\alpha')$, and furthermore because all values of type `void` are equal to `null`, we obtain:

(3.6.37) $\Gamma[\sigma] \vdash \mathcal{E}; (\mathcal{R}^{o})'; s \models H[\sigma] * \texttt{(ex void } \alpha')(\alpha' == \ell * fr' \cdot p.\texttt{postJoin<1>)}$

In addition, applying admissibility of logical consequence (Lemma 6) and (Var Set) to (3.6.8) (like at the end of proof case (Red Fork)), we also get:

(3.6.38) $\Gamma, \Gamma'; r \vdash \begin{array}{c} \{H * \texttt{(ex void } \alpha')(\alpha' == \ell * fr' \cdot p.\texttt{postJoin<1>)}\} \\ \ell = \texttt{null}; c' : \texttt{void} \\ \{fr \cdot o.\texttt{postJoin<1>}\} \end{array}$

Our goal (3.6.35), now follows from (3.6.37) and (3.6.38). □

6.2.3 Data Race Freedom

We obtain data race freedom of verified programs as a corollary of the preservation theorem.

Proof of Theorem 6 (Verified Programs are Data Race Free). *If* $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h,\ ts \mid o_1 \text{ is } (s_1 \text{ in } hc_1; c_1) \mid o_2 \text{ is } (s_2 \text{ in } hc_2; c_2)\rangle$*, then* (hc_1, hc_2) *is not a data race.*

Proof. Let $(ct, c) : \diamond$, $st = \langle h,\ ts \mid o_1 \text{ is } (s_1 \text{ in } hc_1; c_1) \mid o_2 \text{ is } (s_2 \text{ in } hc_2; c_2)\rangle$, and $\mathsf{init}(c) \rightarrow_{ct}^{*} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 13) and preservation (Theorem 3), we know that $st : \diamond$. Suppose, towards a contradiction, that (hc_1, hc_2) is a data race. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be resources $\mathcal{R}$, $\mathcal{R}'$ and a heap cell $o.f$ such that $\mathcal{R} \vdash o_1 \text{ is } (s_1 \text{ in } hc_1; c_1) : \diamond$, $\mathcal{R}' \vdash o_2 \text{ is } (s_2 \text{ in } hc_2; c_2) : \diamond$, $\mathcal{R}\#\mathcal{R}'$, $\mathcal{R}_{\mathsf{perm}}(o, f) = 1$ and $\mathcal{R}'_{\mathsf{perm}}(o, f) > 0$. But then $\mathcal{R}_{\mathsf{perm}}(o, f) + \mathcal{R}'_{\mathsf{perm}}(o, f) > 1$, in contradiction to $\mathcal{R}\#\mathcal{R}'$. □

6.3 Soundness of Chapter 5's Verification System: Preservation

To prove preservation, we need an auxiliary lemma:

Lemma 18 (Monotonicity of Initialized Sets)**.**
If $(\Gamma \vdash \mathcal{E}; \mathcal{R}; s \models F)$, $\mathcal{R}' = (\mathcal{R}_{\mathsf{hp}}, \mathcal{R}_{\mathsf{perm}}, \mathcal{R}_{\mathsf{join}}, \mathcal{R}_{\mathsf{lock}}, \mathcal{R}_{\mathsf{fresh}}, \mathcal{R}_{\mathsf{init}} \cup \mathcal{I})$ *and* $\mathcal{I} \subseteq \mathsf{dom}(\Gamma)$*, then* $(\Gamma \vdash \mathcal{E}; \mathcal{R}'; s \models F)$.

Proof. By induction on the structure of F, making use of the syntactic restriction that the `initialized`-predicate must not occur in negative positions. □

Now, we update Theorem 3's proof to cope with the new cases of the operational semantics. We do not show how existing cases are updated to account for reentrant locks: except in cases (Red New), (Red Fork) and, (Red Join) (where there is little change, because we changed class `Thread`); there is no change at all in Theorem 3's proof.

Proof of Theorem 3 (Preservation). *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$*, then* $(st' : \diamond)$.

Proof.

(1) $ct : \diamond$ assumption
(2) $st : \diamond$ assumption
(3) $st \rightarrow st'$ assumption

An inspection of the reduction rules shows that st is of the following form, where o is the thread that the reduction rule "operates on" (i.e., for all rules but (Red Notify) the only thread on the reduction's left hand side whose components are explicitly named, and for (Red Notify) the thread whose head command is `notify()`):

(4) $st = \langle h,\ l,\ ts \mid o \text{ is } (s \text{ in } c)\rangle$

The proof of $(st : \diamond)$ ends in an application of (State), preceded by an application of (Cons Pool), preceded by an application of (Thread) for thread o. Let $\mathcal{R}^{ts}$ and $\mathcal{R}'$ be the resources that satisfy the thread pool ts and the resource invariants of the initialized, unlocked objects (premises of (Cons Pool) and (State)):

(5) $\mathcal{R}^{ts} \vdash ts : \diamond$
(6) $(\mathcal{R} * \mathcal{R}^{ts})\#\mathcal{R}'$

(7) $l = \mathsf{conc}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})$
(8) $(\mathcal{R}_{\mathsf{free}})_{\mathsf{lock}} = \emptyset$
(9) $\Gamma'' \vdash \mathcal{E}; \mathcal{R}'; \emptyset \models \circledast_{q \in \mathsf{ready}(\mathcal{R} * \mathcal{R}^{ts})} q.\texttt{inv}$

The rule (Thread) includes the following premises:

(10) $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) \subseteq \{o\}$
(11) $\Gamma, \Gamma' \vdash s : \diamond$
(12) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F[\sigma]$
(13) $\Gamma, \Gamma'; r \vdash \{F\}c : \texttt{void}\{G\}$

The proof structure is exactly like in Section 6.2.2. We extend case 3 with the new reduction cases, i.e., (Red Lock), (Red Unlock), (Red No Op) (for`commit`), (Red Wait), (Red Notify), (Red Skip Notify), and (Red Resume). In all these cases, we can assume that the following statements hold:

(14) $c = hc;\ c'$
(15) $\Gamma, \Gamma'; r \vdash \{F\}hc\{F'\}$
(16) $\Gamma, \Gamma'; r \vdash \{F'\}c' : \texttt{void}\{G\}$

We now add cases (Red Lock), (Red Unlock), (Red Wait), (Red Notify), (Red Skip Notify), and (Red Resume) to subcase 3 of Section 6.2.2's proof.

Case 3.7, (Red Lock):

$$\frac{(l(p) = \mathsf{free},\ l' = l[p \mapsto (1, o)]) \text{ or } (l(p) = (n, o),\ l' = l[p \mapsto (n+1, o)])}{\langle h,\ l,\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{lock}();\ c')\rangle \rightarrow \langle h,\ l',\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.7.1) $c = p.\texttt{lock}();\ c'$
(3.7.2) $st' = \langle h,\ l',\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle$
(3.7.3) $(l(p) = \mathsf{free}$ and $l' = l[p \mapsto (1, o)])$ or $(l(p) = (n, o)$ and $l' = l[p \mapsto (n+1, o)])$

The last rules in (13)'s derivation are (Frame) preceded by (Lock) or (Re-Lock). From the premises of these rules, we obtain H and π such that either these judgments hold:

(3.7.4) $F = H * \texttt{Lockset}(\pi) * \texttt{!}(\pi\ \texttt{contains}\ p) * p.\texttt{initialized}$
(3.7.5) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.7.6) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv}\}c' : \texttt{void}\{G\}$

or these judgments hold:

(3.7.7) $F = H * \texttt{Lockset}(p \cdot \pi)$
(3.7.8) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.7.9) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(p \cdot p \cdot \pi)\}c' : \texttt{void}\{G\}$

Now we split cases according to (3.7.3).

Case 3.7.1, $l(p) = \mathsf{free}$ and $l' = l[p \mapsto (1, o)]$: We suppose that (3.7.7), (3.7.8), and (3.7.9) hold and show that this contradicts the assumption that $st : \diamond$. By (12) and (3.7.7), we have:

(3.7.1.1) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H * \texttt{Lockset}(p \cdot \pi)$

By the semantics of $\texttt{Lockset}(p \cdot \pi)$, the previous judgment means that $\mathcal{R}_{\mathsf{lock}}(o)(p) = [\![p \cdot \pi]\!]_s^{\mathcal{R}_{\mathsf{hp}}}(p) \geq 1$. In addition, by statement (7), we have that $l = \mathsf{conc}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})$. From this statement and $\mathcal{R}_{\mathsf{lock}}(o)(p) = [\![p \cdot \pi]\!]_s^{\mathcal{R}_{\mathsf{hp}}} \geq 1$, we deduce (by the second case of conc's definition) that $l(p) = (1, \_)$. This contradicts $l(p) = \mathsf{free}$: this case is impossible. Therefore, we can suppose that (3.7.4), (3.7.5), and (3.7.6) hold.

From (12) and (3.7.4), we have:

(3.7.1.2) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H * \texttt{Lockset}(\pi) * \texttt{!}(\pi\ \texttt{contains}\ p) * p.\texttt{initialized}$

From (3.7.1.2), by the semantics of $p.\texttt{initialized}$, it follows that $p \in \mathcal{R}_{\mathsf{init}}$. By the way initialized sets are joined, it follows that $p \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$. In addition we have $l(p) = \mathsf{free}$. From the previous statement and $l = \mathsf{conc}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})$ (statement (7)) it follows that $(\forall q \in \mathsf{dom}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}))(p \notin (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}(q))$ holds (by the first case of conc's definition). From the previous statement and $p \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$, it follows that $p \in \mathsf{ready}(\mathcal{R} * \mathcal{R}_{ts})$. Therefore, by (9) and case $*$ of the formula semantics, there exist $\mathcal{R}^p$ and $\mathcal{R}'_{\mathsf{free}}$ such that:

(3.7.1.3) $\mathcal{R}_{\mathsf{free}} = \mathcal{R}^p * \mathcal{R}'_{\mathsf{free}}$
(3.7.1.4) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^p; \emptyset \models p.\texttt{inv}$
(3.7.1.5) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'_{\mathsf{free}}; \emptyset \models \circledast_{q \in (\mathsf{ready}(\mathcal{R} * \mathcal{R}_{ts}) \setminus p)} q.\texttt{inv}$

We define: $\mathcal{R}^\natural \triangleq (\mathcal{R}_{\mathsf{hp}}, \mathcal{R}_{\mathsf{perm}}, \mathcal{R}_{\mathsf{join}}, \mathcal{R}_{\mathsf{lock}}[o \mapsto (\mathcal{R}_{\mathsf{lock}}(o) \sqcup [p])], \mathcal{R}_{\mathsf{fresh}}, \mathcal{R}_{\mathsf{init}})$. By case $\texttt{Lockset}(p \cdot \pi)$ of the formula semantics and (3.7.1.2), it follows that:

(3.7.1.6) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^\natural; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi)$

We define: $\mathcal{R}' \triangleq \mathcal{R}^\natural * \mathcal{R}^p$. Because $\mathcal{R} \# \mathcal{R}_{ts} \# \mathcal{R}_{\mathsf{free}}$ and $\mathcal{R}^p \leq \mathcal{R}_{\mathsf{free}}$ (this follows from $\mathcal{R}^p * \mathcal{R}'_{\mathsf{free}} = \mathcal{R}_{\mathsf{free}}$), we have $\mathcal{R} \# \mathcal{R}^p$. In addition, because $(\mathcal{R}_{\mathsf{free}})_{\mathsf{lock}} = \emptyset$ (statement (8)), we have that $o \notin \mathsf{dom}((\mathcal{R}^p)_{\mathsf{lock}})$. Putting this all together we obtain:

(3.7.1.7) $\mathcal{R}^\natural \# \mathcal{R}^p$

From (3.7.1.6), (3.7.1.4), (3.7.1.7), and case $*$ of the formula semantics, it follows that:

(3.7.1.8) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv}$

We apply (Thread) to (3.7.1.8) and (3.7.6). We obtain that the thread o is verified: $(\mathcal{R}' \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c') : \diamond)$. We apply (Cons Pool) to this judgment and (5). We obtain:

(3.7.1.9) $\mathcal{R}' * \mathcal{R}_{ts} \vdash t \mid ts : \diamond$

Now we establish that resource axiom (b) is preserved for resource $\mathcal{R}' * \mathcal{R}_{ts}$ (the proof is similar for $\mathcal{R}'$):

(3.7.1.10) $\{q \mid q \in (\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}}(\_)\} \subseteq (\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{init}}$ (goal)

By the definition of $\mathcal{R}'$, we have:

(3.7.1.11) $\{q \mid q \in (\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}}(\_)\} = \{q \mid q \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}(\_)\} \cup \{p\}$

Because $p \in \mathcal{R}_{\mathsf{init}}$ (this follows from (3.7.1.2)), by the way initialized sets are joined, we have $p \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$. Now $(\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{init}} = (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$, equality (3.7.1.11) and resource property (b) for $\mathcal{R} * \mathcal{R}_{ts}$ suffice to show (3.7.1.10).

From the definitions of $\mathcal{R}'$, $p\ \texttt{!=}\ \texttt{null}$ and l', it follows that $l' = \mathsf{conc}((\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}})$. From this judgment, $\mathcal{R}' \# \mathcal{R}_{ts} \# \mathcal{R}'_{\mathsf{free}}$ (this follows from the definitions of $\mathcal{R}'$, $\mathcal{R}_{ts}$, and $\mathcal{R}'_{\mathsf{free}}$, and $\mathcal{R} \# \mathcal{R}_{ts} \# \mathcal{R}_{\mathsf{free}}$), (3.7.1.5), and (3.7.1.9), we can apply (State). We obtain $st' : \diamond$.

Case 3.7.2, $l(p) = (n, o)$ and $l' = l[p \mapsto (n+1, o)]$: We suppose that (3.7.4), (3.7.5), and (3.7.6) hold and show that this contradicts the assumption that $st : \diamond$. By (12) and (3.7.4), we have:

(3.7.2.1) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(\pi) * !(\pi \texttt{ contains } p) * p.\texttt{initialized}$

By the semantics of $!(\pi \texttt{ contains } p)$, the previous judgment means that $p \notin \mathcal{R}_{\mathsf{lock}}(o)$ and $[\![\pi]\!]_s^{\mathcal{R}_{\mathsf{hp}}}(p) = 0$. In addition, by statement (7), we have that $l = \mathsf{conc}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})$. From this statement and $p \notin \mathcal{R}_{\mathsf{lock}}(o)$, we deduce (by the first case of conc's definition) that $l(p) = \mathsf{free}$ or (by the second case of conc's definition) that $l(p) = (\_, q)$ with $q \neq o$. In both cases, this contradicts $l(p) = (n, o)$: this case is impossible. Therefore, we can suppose that (3.7.7), (3.7.8), and (3.7.9) hold.

Like before, from (12) and (3.7.7), we have:

(3.7.2.2) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi)$

By the semantics of `lockset`, we have $p \in \mathcal{R}_{\mathsf{lock}}(o)$. We define:

(3.7.2.3) $\mathcal{R}' \triangleq (\mathcal{R}_{\mathsf{hp}}, \mathcal{R}_{\mathsf{perm}}, \mathcal{R}_{\mathsf{join}}, \mathcal{R}_{\mathsf{lock}}[o \mapsto (\mathcal{R}_{\mathsf{lock}}(o) \sqcup [p])], \mathcal{R}_{\mathsf{fresh}}, \mathcal{R}_{\mathsf{init}})$.

By the semantics of $\texttt{Lockset}(p \cdot p \cdot \pi)$ and (3.7.2.2) it follows that:

(3.7.2.4) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(p \cdot p \cdot \pi)$

We apply (Thread) to (3.7.2.4) and (3.7.9). We obtain that the thread o is verified: $(\mathcal{R}' \vdash o \mathsf{\ is\ } (s \mathsf{\ in\ } c') : \diamond)$. We apply (Cons Pool) to this judgment and (5) to obtain that the thread pool is verified $(\mathcal{R}' * \mathcal{R}_{ts} \vdash t \mid ts : \diamond)$. From this judgment and $l' = \mathsf{conc}((\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}})$ (this follows from (7) and the definitions of $\mathcal{R}'$ and l') we can apply (State). We obtain $st' : \diamond$.

Case 3.8, (Red Unlock):

$$\frac{l(p) = (n, o) \quad n = 1 \Rightarrow l' = l[p \mapsto \mathsf{free}] \quad n > 1 \Rightarrow l' = l[p \mapsto (n-1, o)]}{\langle h,\ l,\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } p.\texttt{unlock}();\ c')\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.8.1) $c = p.\texttt{unlock}();\ c'$
(3.8.2) $st' = \langle h,\ l',\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } c')\rangle$
(3.8.3) $(n = 1$ and $l' = l[p \mapsto \mathsf{free}])$ or $(n > 1$ and $l' = l[p \mapsto (n-1, o)])$

The last rules in (13)'s derivation are (Frame) preceded by (Unlock) or (Re-Unlock). From the premises of these rules, we obtain H and π such that either these judgments hold:

(3.8.4) $F = H * \texttt{Lockset}(p \cdot p \cdot \pi)$
(3.8.5) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.8.6) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(p \cdot \pi)\} c' : \texttt{void}\{G\}$

or these judgments hold:

(3.8.7) $F = H * \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv}$
(3.8.8) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.8.9) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(\pi)\} c' : \texttt{void}\{G\}$

Now we do a case split on (3.8.3).

Case 3.8.1, $n = 1$ and $l' = l[p \mapsto \mathsf{free}]$: We suppose that (3.8.4), (3.8.5), and (3.8.6) hold and show that this contradicts the assumption that $st : \diamond$. By (12) and using (3.8.4), we have:

(3.8.1.1) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(p \cdot p \cdot \pi)$

By the semantics of $\texttt{Lockset}(p \cdot p \cdot \pi)$, the previous judgment means that $\mathcal{R}_{\mathsf{lock}}(o)(p) = [\![p \cdot p \cdot \pi]\!]_s^{\mathcal{R}_{\mathsf{hp}}}(p) \geq 2$. In addition, by statement (7), we have that $l = \mathsf{conc}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})$. From this statement and $\mathcal{R}_{\mathsf{lock}}(o)(p) = [\![p \cdot p \cdot \pi]\!]_s^{\mathcal{R}_{\mathsf{hp}}}(p) \geq 2$, we deduce (by the second case of conc's definition) that $l(p) = (2, \_)$. This contradicts $l(p) = (1, o)$: this case is impossible. Therefore, we can suppose that (3.8.7), (3.8.8), and (3.8.9) hold.

By (12) and (3.8.7), we have:

(3.8.1.2) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi) * p.\texttt{inv}$

From this judgment, by the semantics of $*$, it follows that there exist $\mathcal{R}^1$ and $\mathcal{R}^2$ such that the following judgments hold:

(3.8.1.3) $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$
(3.8.1.4) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^1; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi)$
(3.8.1.5) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^2; s \models p.\texttt{inv}$

We define (note that the operator $\setminus$ is overloaded for bags: it removes one occurence of its right hand side from its left hand side):

(3.8.1.6) $\mathcal{R}' \triangleq (\mathcal{R}^1_{\mathsf{hp}}, \mathcal{R}^1_{\mathsf{perm}}, \mathcal{R}^1_{\mathsf{join}}, \mathcal{R}^1_{\mathsf{lock}}[o \mapsto (\mathcal{R}^1_{\mathsf{lock}}(o) \setminus p)], \mathcal{R}^1_{\mathsf{fresh}}, \mathcal{R}^1_{\mathsf{init}})$

By (3.8.1.4) and the definition of $\mathcal{R}'$, it follows that:

(3.8.1.7) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(\pi)$

We apply (Thread) to the previous judgment and (3.8.9) to obtain that thread o is verified: $(\mathcal{R}' \vdash o \ \mathsf{is}\ (s \ \mathsf{in}\ c') : \diamond)$.
Now we establish the invariant that there is an $\mathcal{R}'_{\mathsf{free}}$ that satisfies the $*$-conjunction of monitor invariants of unheld locks (i.e., we reestablish assumption (9)) (note that $p \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$ holds because of resource axiom (b) for $\mathcal{R} * \mathcal{R}_{ts}$ and because $\mathcal{R}_{\mathsf{lock}}(o)(p) = 1$). We define:

(3.8.1.8) $\mathcal{R}'_2 \triangleq (\mathcal{R}^2_{\mathsf{hp}}, \mathcal{R}^2_{\mathsf{perm}}, \mathcal{R}^2_{\mathsf{join}}, \emptyset, \mathcal{R}^2_{\mathsf{fresh}}, \mathcal{R}^2_{\mathsf{init}})$
(3.8.1.9) $\mathcal{R}'_{\mathsf{free}} \triangleq \mathcal{R}_{\mathsf{free}} * \mathcal{R}'_2$

By (10), we have $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) \subseteq \{o\}$. Further, by $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$ and (3.8.1.4), we have $\mathsf{dom}(\mathcal{R}^1) = \{o\}$. Because $\mathcal{R}^1 \# \mathcal{R}^2$, it follows that $\mathcal{R}^2_{\mathsf{lock}} = \emptyset$. From the previous statement, it follows that we can set the lock table to the empty set in (3.8.1.5). We obtain:

(3.8.1.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'_2; s \models p.\texttt{inv}$

From the definition of $\mathcal{R}'$, the previous judgment, (9), and (8), it follows that:

(3.8.1.11) $\mathsf{ready}(\mathcal{R}' * \mathcal{R}'_{ts}) = \mathsf{ready}(\mathcal{R} * \mathcal{R}'_{ts}) \cup \{p\}$
(3.8.1.12) $\Gamma \vdash \mathcal{E}; \mathcal{R}'_{\mathsf{free}}; \emptyset \models \circledast_{q \in \mathsf{ready}(\mathcal{R}' * \mathcal{R}'_{ts})} q.\texttt{inv}$
(3.8.1.13) $(\mathcal{R}'_{\mathsf{free}})_{\mathsf{lock}} = \emptyset$

Resource property (b) for $\mathcal{R}' * \mathcal{R}_{ts}$ is preserved because of (1) resource property (b) for $\mathcal{R} * \mathcal{R}_{ts}$ and (2) the following statement:

$$\{q \mid (\forall q' \in \mathsf{dom}((\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}}))(q \notin (\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}})(q')\} \subset \{q \mid (\forall q' \in \mathsf{dom}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}))(q \notin (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}})(q')\} \tag{3.8.10}$$

Therefore, from $l' = \mathsf{conc}(\mathcal{R}' * \mathcal{R}_{ts})$ (this follows from (7) and the definition of $\mathcal{R}'$), $(\mathcal{R}' \vdash o \ \mathsf{is}\ (s \ \mathsf{in}\ c') : \diamond)$, (3.8.1.12), $\mathcal{R}' \# \mathcal{R}_{ts} \# \mathcal{R}'_{\mathsf{free}}$ (this follows from (6)), and (3.8.1.13), we can apply (State). We obtain: $st' : \diamond$.

Case 3.8.2, $n > 1$ and $l' = l[p \mapsto (n-1, o)]$: We suppose that (3.8.4), (3.8.5), and (3.8.6) hold. We omit the proof in case where (3.8.7), (3.8.8), and (3.8.9) hold because it is simpler (the hypotheses are stronger). By (3.8.4) and (12), we have:

(3.8.2.1) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(p \cdot p \cdot \pi)$

We define:

(3.8.2.2) $\mathcal{R}' \triangleq (\mathcal{R}_{\mathsf{hp}}, \mathcal{R}_{\mathsf{perm}}, \mathcal{R}_{\mathsf{join}}, \mathcal{R}_{\mathsf{lock}}[o \mapsto (\mathcal{R}_{\mathsf{lock}}(o) \setminus p)], \mathcal{R}_{\mathsf{fresh}}, \mathcal{R}_{\mathsf{init}})$

By the definition of $\mathcal{R}'$ and (3.8.2.1) it follows that:

(3.8.2.3) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(p \cdot \pi)$

We apply (Thread) to the previous judgment and (3.8.6). We obtain that thread o is verified:

(3.8.2.4) $(\mathcal{R}' \vdash o \ \mathsf{is}\ (s \ \mathsf{in}\ c') : \diamond)$

We apply (Cons Pool) and (State) to $l' = \mathsf{conc}(\mathcal{R}' * \mathcal{R}_{ts})$ (this follows from (7) and the definition of $\mathcal{R}'$), (3.8.2.4) and (5) (note that invariant (9) is preserved because $\mathsf{ready}(\mathcal{R} * \mathcal{R}_{ts}) = \mathsf{ready}(\mathcal{R}' * \mathcal{R}_{ts})$). We obtain: $st' : \diamond$.

Case 3.9, (Red No Op) for `commit`:

$$\frac{}{\langle h,\ l,\ ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ p.\texttt{commit}; c')\rangle \rightarrow \langle h,\ l,\ ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.9.1) $c = p.\texttt{commit}; c'$
(3.9.2) $st' = \langle h,\ l,\ ts \mid o \ \mathsf{is}\ (s \ \mathsf{in}\ c')\rangle$

The last rules in (13)'s derivation are (Frame) preceded by (Commit). From the rule premises we obtain H and π such that the following statements hold:

(3.9.3) $F = H * \texttt{Lockset}(\pi) * p.\texttt{inv} * p.\texttt{fresh}$
(3.9.4) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.9.5) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(\pi) * !(\pi \ \texttt{contains}\ p) * p.\texttt{initialized}\} c' : \texttt{void}\{G\}$

By (13) and (3.9.3), we have:

(3.9.6) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models H[\sigma] * \texttt{Lockset}(\pi) * p.\texttt{inv} * p.\texttt{fresh}$

From the previous judgment, by the semantics of $*$, it follows that there exist $\mathcal{R}^1$ and $\mathcal{R}^2$ such that the following judgments hold:

(3.9.7) $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$
(3.9.8) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^1; s \models H[\sigma] * \texttt{Lockset}(\pi)$
(3.9.9) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^2; s \models p.\texttt{inv} * p.\texttt{fresh}$

By judgment (3.9.6) and the semantics of $p.\texttt{fresh}$, it follows that $p \in \mathcal{R}_{\mathsf{fresh}}$. In addition, by resource property (a), we have $(\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{fresh}} \cap (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}} = \emptyset$. From the way resources are joined, it follows that $p \notin \mathcal{R}_{\mathsf{init}}$. In addition, by $p \in \mathcal{R}_{\mathsf{fresh}}$, judgment (3.9.9), and the way fresh sets are joined, we have $p \notin (\mathcal{R}_{ts})_{\mathsf{fresh}}$. Putting this all together, we can define:

(3.9.10) $\mathcal{R}' \triangleq (\mathcal{R}^1_{\mathsf{hp}}, \mathcal{R}^1_{\mathsf{perm}}, \mathcal{R}^1_{\mathsf{join}}, \mathcal{R}^1_{\mathsf{lock}}, \mathcal{R}^1_{\mathsf{fresh}} \setminus p, \mathcal{R}^1_{\mathsf{init}} \cup \{p\})$

(3.9.11) $\mathcal{R}'_{ts} \triangleq ((\mathcal{R}_{ts})_{\mathsf{hp}}, (\mathcal{R}_{ts})_{\mathsf{loc}}, (\mathcal{R}_{ts})_{\mathsf{join}}, (\mathcal{R}_{ts})_{\mathsf{lock}}, (\mathcal{R}_{ts})_{\mathsf{fresh}}, (\mathcal{R}_{ts})_{\mathsf{init}} \cup \{p\})$

From the definitions of $\mathcal{R}'$ and $\mathcal{R}'_{ts}$, and $\mathcal{R} \# \mathcal{R}_{ts}$ (statement (6)), it follows that:

(3.9.12) $\mathcal{R}' \# \mathcal{R}'_{ts}$

By resource property (b) for $(\mathcal{R} * \mathcal{R}_{ts})$ and $p \in \mathcal{R}_{\mathsf{fresh}}$, by contradiction, we have:

(3.9.13) $(\forall q \in \mathsf{dom}((\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}))(p \notin (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}(q))$

By the definition of $\mathcal{R}'$, we have $p \in \mathcal{R}'_{\mathsf{init}}$. By the semantics of $p.\texttt{initialized}$ and judgment (3.9.8), it follows that:

(3.9.14) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(\pi) * p.\texttt{initialized}$

Let $h^\natural = \mathcal{R}_{\mathsf{hp}}$. By statement (3.9.13) and judgment (3.9.6), it follows that that $[\![\pi]\!]^{h^\natural}_s(p) = 0$ and $p \notin \mathcal{R}_{\mathsf{lock}}(o)$. By the semantics of $\texttt{!}(\pi\ \texttt{contains}\ p)$, and $\mathcal{R}'_{\mathsf{lock}} = \mathcal{R}_{\mathsf{lock}}$, it follows that:

(3.9.15) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H[\sigma] * \texttt{Lockset}(\pi) * \texttt{!}(\pi\ \texttt{contains}\ p) * p.\texttt{initialized}$

We apply (Thread) to the previous judgment and (3.9.5). We obtain that thread o is verified: $(\mathcal{R}' \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c') : \diamond)$. By Lemma 18 and statement (5), it follows that we can apply (Cons Pool) to obtain that the thread pool is verified: $(\mathcal{R}'_{ts} \vdash ts : \diamond)$. We apply (Cons Pool) to the previous statement and $(\mathcal{R}' \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ c') : \diamond)$. We obtain:

(3.9.16) $\mathcal{R}' * \mathcal{R}'_{ts} \vdash o \mid ts : \diamond$

Now we establish the invariant that there is an $\mathcal{R}'_{\mathsf{free}}$ that satisfy the $*$-conjunction of monitor invariants of unheld locks (invariant (9)). We define:

(3.9.17) $\mathcal{R}^\natural_{\mathsf{free}} \triangleq ((\mathcal{R}_{\mathsf{free}})_{\mathsf{hp}}, (\mathcal{R}_{\mathsf{free}})_{\mathsf{loc}}, (\mathcal{R}_{\mathsf{free}})_{\mathsf{join}}, (\mathcal{R}_{\mathsf{free}})_{\mathsf{lock}}, (\mathcal{R}_{\mathsf{free}})_{\mathsf{fresh}}, (\mathcal{R}_{\mathsf{free}})_{\mathsf{init}} \cup \{p\})$

(3.9.18) $\mathcal{R}'_2 \triangleq (\mathcal{R}^2_{\mathsf{hp}}, \mathcal{R}^2_{\mathsf{perm}}, \mathcal{R}^2_{\mathsf{join}}, \mathcal{R}^2_{\mathsf{fresh}}, \emptyset, \mathcal{R}^2_{\mathsf{init}} \cup \{p\})$

(3.9.19) $\mathcal{R}'_{\mathsf{free}} \triangleq \mathcal{R}^\natural_{\mathsf{free}} * \mathcal{R}'_2$

By Lemma 18, we can add p to the resource's initialized set of judgment (3.9.9). By (10), we have $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) \subseteq \{o\}$. Further, by $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$ and (3.9.8), we have $\mathsf{dom}(\mathcal{R}^1) = \{o\}$. Because $\mathcal{R}^1 \# \mathcal{R}^2$, it follows that $\mathcal{R}^2_{\mathsf{lock}} = \emptyset$. From the previous statement, it follows that we can set the lock table to the empty set in judgment (3.9.9). We obtain:

(3.9.20) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'_2; s \models p.\texttt{inv}$

By (3.9.13) and $p \in (\mathcal{R}' * \mathcal{R}'_{ts})_{\mathsf{init}}$, we have:

(3.9.21) $\mathsf{ready}(\mathcal{R}' * \mathcal{R}'_{ts}) = \mathsf{ready}(\mathcal{R} * \mathcal{R}_{ts}) \cup \{p\}$

From (3.9.20), (9), (3.9.21), and (8), it follows that:

(3.9.22) $\Gamma \vdash \mathcal{E}; \mathcal{R}'_{\mathsf{free}}; \emptyset \models \circledast_{o \in \mathsf{ready}(\mathcal{R}' * \mathcal{R}'_{ts})} o.\texttt{inv}$

(3.9.23) $(\mathcal{R}'_{\mathsf{free}})_{\mathsf{lock}} = \emptyset$

Now, we show that resource property (a) for $\mathcal{R}' * \mathcal{R}'_{ts}$ is preserved i.e., we show:

(3.9.24) $(\mathcal{R}' * \mathcal{R}'_{ts})_{\mathsf{fresh}} \cap (\mathcal{R}' * \mathcal{R}'_{ts})_{\mathsf{init}} = \emptyset$ (goal)

By resource property (a) for $\mathcal{R} * \mathcal{R}_{ts}$, we have that:

(3.9.25) $(\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{fresh}} \cap (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}} = \emptyset$

In addition, by the definitions of $\mathcal{R}'$ and $\mathcal{R}'_{ts}$, we have:

(3.9.26) $(\mathcal{R}' * \mathcal{R}'_{ts})_{\mathsf{fresh}} = (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{fresh}} \setminus p$
(3.9.27) $(\mathcal{R}' * \mathcal{R}'_{ts})_{\mathsf{init}} = (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}} \cup \{p\}$

Now, goal (3.9.24) follows from (3.9.25), (3.9.26), and (3.9.27).

From (3.9.16), $\mathcal{R}' \# \mathcal{R}'_{ts} \# \mathcal{R}'_{\mathsf{free}}$ (this follows from (6)), (3.9.23), and (3.9.22), we can apply (State). We obtain: $st' : \diamond$.

Case 3.10, (Red Wait):

$$\frac{l(p) = (n, o) \quad l' = l[p \mapsto \mathsf{free}]}{\langle h,\ l,\ ts \mid o \ \mathsf{is}\ (s\ \mathsf{in}\ \ell = p.\texttt{wait}();\ c')\rangle \ \rightarrow\ \langle h,\ l',\ ts \mid o \ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{waiting}(n);\ p.\texttt{resume}(n);\ c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.10.1) $c = p.\texttt{wait}()\ c'$
(3.10.2) $st' = \langle h,\ l',\ ts \mid o \ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{waiting}(n);\ p.\texttt{resume}(n);\ c')\rangle$
(3.10.3) $l(p) = (n, o)$

The last rules in (13)'s derivation are (Frame) preceded by (Call). By method `wait`'s specification in class `Thread`, it follows that; from the premises of rules (Frame) and (Call), we obtain H and π such that these judgments hold:

(3.10.4) $F = H * \texttt{Lockset}(\pi) * \pi\ \texttt{contains}\ p * p.\texttt{inv}$
(3.10.5) $\Gamma, \Gamma' \vdash p, \pi, H : \texttt{Object}, \texttt{lockset}, \diamond$
(3.10.6) $\Gamma, \Gamma'; r \vdash \{H * \texttt{Lockset}(\pi) * p.\texttt{inv}\} c' : \texttt{void}\{G\}$

From (13) and (3.10.4), it follows that:

(3.10.7) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}; s \models F' * \texttt{Lockset}(e) * e\ \texttt{contains}\ p * p.\texttt{inv}$

Let π' be the multiset such that $p^n \cdot \pi' = \pi$ and such that π' does not contain p. Note that π' exists by (3.10.7) and the semantics of the `Lockset` predicate. By (3.10.6), we can apply (Waiting) and (Resume) to $p.\texttt{waiting}(n);\ p.\texttt{resume}(n);\ c'$. We obtain:

(3.10.8) $\Gamma[\sigma]; v \vdash \begin{array}{c}\{H * \texttt{Lockset}(\pi') * !(\pi'\ \texttt{contains}\ p) * p.\texttt{initialized}\} \\ p.\texttt{waiting}(n);\ p.\texttt{resume}(n);\ c' : \texttt{void} \\ \{G\}\end{array}$

By (3.10.7) and the semantics of $*$, it follows that there exist $\mathcal{R}^1$ and $\mathcal{R}^2$ such that the following statements hold:

(3.10.9) $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$
(3.10.10) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^1; s \models H * \texttt{Lockset}(\pi) * \pi\ \texttt{contains}\ p$
(3.10.11) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}^2; s \models p.\texttt{inv}$

Let $h^\natural = \mathcal{R}^1_{\mathsf{hp}}$. By (3.10.10), it follows that $\mathcal{R}^1_{\mathsf{lock}}(p)(o) = [\![\pi]\!]_s^{h^\natural}(p) \geq 1$. By the way abstract lock tables are composed, it follows that $(\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{lock}}(p)(o) \geq 1$. Because of resource property (b) for $\mathcal{R} * \mathcal{R}_{ts}$, it follows that $p \in (\mathcal{R} * \mathcal{R}_{ts})_{\mathsf{init}}$. By the way initialized sets are joined, it follows that:

(3.10.12) $p \in \mathcal{R}^1_{\mathsf{init}}$

We define (where $\backslash\backslash$ is an operator that removes all occurences of its right hand side from its left hand side):

(3.10.13) $\mathcal{R}' \triangleq (\mathcal{R}^1_{\mathsf{hp}}, \mathcal{R}^1_{\mathsf{perm}}, \mathcal{R}^1_{\mathsf{join}}, \mathcal{R}^1_{\mathsf{lock}}[o \mapsto (\mathcal{R}^1_{\mathsf{lock}}(o) \backslash\backslash p)], \mathcal{R}^1_{\mathsf{fresh}}, \mathcal{R}^1_{\mathsf{init}})$

(3.10.14) $\mathcal{R}'' \triangleq (\mathcal{R}^2_{\mathsf{hp}}, \mathcal{R}^2_{\mathsf{perm}}, \mathcal{R}^2_{\mathsf{join}}, \emptyset, \mathcal{R}^2_{\mathsf{fresh}}, \mathcal{R}^2_{\mathsf{init}})$

By (3.10.10), (3.10.12), the definitions of $\mathcal{R}'$ and π', the semantics of `Lockset(`π'`)` and `!(`π' `contains` p`)`, it follows that:

(3.10.15) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}'; s \models H * \texttt{Lockset(}\pi'\texttt{)} * \texttt{!(}\pi' \texttt{ contains } p\texttt{)} * p\texttt{.initialized}$

We apply (Thread) to (3.10.15) and (3.10.8). We obtain that the thread o is verified:

(3.10.16) $\mathcal{R}' \vdash o \mathsf{\ is\ } (s \mathsf{\ in\ } p\texttt{.waiting(}n\texttt{)}; p\texttt{.resume(}n\texttt{)}; c') : \diamond$

We apply (Cons Pool) to the previous judgment and (5). We obtain that the thread pool is verified: $(\mathcal{R}' * \mathcal{R}_{ts} \vdash o \mid ts : \diamond)$.

Now we establish the invariant that there is an $\mathcal{R}'_{\mathsf{free}}$ that satisfy the $*$-conjunction of monitors invariants of unheld locks (i.e., that invariant (9) is preserved).

By (10), we have $\mathsf{dom}(\mathcal{R}_{\mathsf{lock}}) \subseteq \{o\}$. Further, by $\mathcal{R} = \mathcal{R}^1 * \mathcal{R}^2$ and (3.10.10), we have $\mathsf{dom}(\mathcal{R}^1) = \{o\}$. Because $\mathcal{R}^1 \# \mathcal{R}^2$, it follows that $\mathcal{R}^2_{\mathsf{lock}} = \emptyset$. From the previous statement, it follows that we can set the abstract lock table to the empty set in judgment (3.10.11). We obtain:

(3.10.17) $\Gamma[\sigma] \vdash \mathcal{E}; \mathcal{R}''; s \models p\texttt{.inv}$

Because $(\mathcal{R} * \mathcal{R}_{ts}) \# \mathcal{R}_{\mathsf{free}}$ (judgment (6)), we have $\mathcal{R}'' \# \mathcal{R}_{\mathsf{free}}$. We define:

(3.10.18) $\mathcal{R}'_{\mathsf{free}} \triangleq \mathcal{R}'' * \mathcal{R}_{\mathsf{free}}$

By the definition of $\mathcal{R}'$ (recall that $p \notin (\mathcal{R}')_{\mathsf{lock}}(o)$), the way abstract lock tables are joined, and $\mathcal{R}^1_{\mathsf{lock}}(p)(o) > 0$, it follows that $\mathsf{ready}(\mathcal{R}' * \mathcal{R}_{ts}) = \mathsf{ready}(\mathcal{R} * \mathcal{R}_{ts}) \cup \{p\}$. From the previous judgment, it follows that we can combine (3.10.17) and (9) by case $*$ of the formula semantics to obtain:

(3.10.19) $\Gamma \vdash \mathcal{E}; \mathcal{R}'_{\mathsf{free}}; \emptyset \models \circledast_{o \in \mathsf{ready}(\mathcal{R}' * \mathcal{R}_{ts})} o\texttt{.inv}$

Because $(\mathcal{R}' * \mathcal{R}_{ts}) \# \mathcal{R}'_{\mathsf{free}}$ (this follows from (6)), $l' = \mathsf{conc}((\mathcal{R}' * \mathcal{R}_{ts})_{\mathsf{lock}})$ (this follows from (7) and the definitions of $\mathcal{R}'$ and l'), $(\mathcal{R}'_{\mathsf{free}})_{\mathsf{lock}} = \emptyset$ (this follows from the definition of $\mathcal{R}''$ and (8)), we can apply (State) to $(\mathcal{R}' * \mathcal{R}_{ts} \vdash o \mid ts : \diamond)$. We obtain: $st' : \diamond$.

Case 3.11, (Red Notify):

$$\frac{l(p) = (n, o)}{\langle h,\ l,\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } \ell = p\texttt{.notify(); } c') \mid q \mathsf{\ is\ } (s_q \mathsf{\ in\ } p\texttt{.waiting(}n'\texttt{); } c_q)\rangle \rightarrow \langle h,\ l,\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } c') \mid q \mathsf{\ is\ } (s_q \mathsf{\ in\ } c_q)\rangle}$$

This proof case is trivial, because in the specification of `notify` and the Hoare rule for `waiting`, the precondition implies the postcondition.

Case 3.12, (Red Skip Notify):

$$\frac{l(p) = (n, o)}{\langle h,\ l,\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } \ell = p\texttt{.notify(); } c')\rangle \rightarrow \langle h,\ l,\ ts \mid o \mathsf{\ is\ } (s \mathsf{\ in\ } c')\rangle}$$

This proof case is trivial, because `notify`'s precondition implies its postcondition.

Case 3.13, (Red Resume):

$$\frac{l(p) = \mathsf{free} \quad l' = l[p \mapsto (n, o)]}{\langle h,\ l,\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ p.\texttt{resume(}n\texttt{)};\ c')\rangle \rightarrow \langle h,\ l',\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle}$$

This proof case is very similar to the proof case for (Red Lock).

□

Chapter 7

Specifying Protocols and Checking Method Contracts

Classes and interfaces are often developed with a particular usage pattern in mind, i.e., methods are supposed to be called in a particular order. Such a usage pattern can be described with a *protocol*. Protocols provide a higher level of specification, compared to method contracts in the previous chapters. In this chapter, we show how to (1) specify protocols (i.e., allowed sequences of method calls) of classes and (2) how to check that method contracts are correct w.r.t. protocols.

This chapter is structured as follows: In Section 7.1 we informally explain what we mean by protocols, in Section 7.2 we present the formal language we use to specify protocols, and in Section 7.3 we illustrate this by specifying the protocols of Section 3.6.2's Iterator and Section 3.6.1's Roster. In Section 7.4 we define the semantics of protocols and in Section 7.5 we present a new technique to check that method contracts are correct w.r.t. protocols.

Note: The work presented in this chapter is published in the ACM Symposium on Applied Computing (SAC 2009) [64].

7.1 Protocols: Introduction

Classes and interfaces must sometimes be used according to a given protocol, i.e., method calls should be done in a given order. Java's `StreamBuffer` is an example of an interface that must be used in a particular way : clients of this interface must call method `read()` zero or more times and then call method `close()` *once* [80]. This can be concisely specified with the following protocol:

`read()*, close()` (StreamBuffer)

More generally, a protocol is a regular expression indicating in which order methods must be called. In this chapter, we extend Cheon et al.'s work [32] to deal with a variant of parameterized and multithreaded classes (i.e., classes such that multiple threads can safely execute in parallel on instances of these classes). We extend the specification language and the semantical foundations of the work mentioned above. We maintain

Cheon et al.'s spirit by providing a concise and intuitive regular expression-like notation to write protocols of multithreaded programs.

In addition, we provide a technique to check that method contracts are correct w.r.t. protocols. For example, given the protocol (StreamBuffer) above, a programmer has to make sure that (1) `read()`'s postcondition implies `reads`'s precondition (because `read` can be called multiple times successively) and (2) `read()`'s postcondition implies `close()`'s precondition (because `close` is called after `read`). If one of the conditions above does not hold, some programs, even if they obey `StreamBuffer`'s protocol, will fail to verify.

Our approach permits to check that method contracts are correct w.r.t. protocols before implementing methods. According to the design by contract approach [79], a good development process follows the following steps: (1) write high-level relations between classes (UML-style specifications), (2) write high-level properties of individual classes (protocols, invariants), (3) write method contracts, and (4) implement methods. However, in the real world, most tools for software verification [6, 35] (including the verification system described in previous chapters) are useful to check implementation of methods against their contracts: as a result, the feedback from software verification tools comes very late in the development process. A strength of our approach is to allow the use of static checking early in the development process, which is – as outlined above – a good practice for software development.

Consequently, our approach makes it easier to write correct specifications, because it makes sure that the method level specifications respect the (more intuitive, simpler to write) high-level protocol specifications.

We believe that support to write correct specifications (as our technique provides) is crucial to encourage programmers to use formal methods. Our claim is supported by the fact that it is easier to write and debug programs than to write and debug specifications. For example, to test a program, one simply has to run it, whereas to test specifications one needs tool support (such as JML's runtime assertion checker [31]). In addition, we believe writing specifications in separation logic is harder than writing specifications in first order logic (such as JML [28] or Boogie [6]). When writing separation logic specifications, one has to specify the threads's permissions to access the heap and the functional behavior of the program, whereas first order logic focuses solely on expressing the functional behavior of the program.

7.2 Syntax

Protocols are specified by the grammar below (where we overload meta-variable s to range over protocols). Section 3.2.1's expressions are extended to include protocol variables.

$$
\begin{array}{rll}
w \in \mathsf{ProtVar} & \text{protocol variables} & \\
e \in \mathsf{Exp} & ::= & \pi \mid \ell \mid w \mid op(\bar{e}) \\
s \in \mathsf{ProtSpec} & ::= & m \mid w = m \mid s,\ s \mid s \mid s \\
& \mid & s? \mid s* \mid s+ \mid e\ ?\ s : s \mid s \mid\mid s \mid \texttt{!<}n\texttt{>}\ s
\end{array}
$$

Protocol specifications are single method specifications (whose return value is possibly assigned to a variable ($w = m$)), or sequential composition of two protocol specifications ($s,\ s$), or a composition of other protocols specifications with regular expressions operators (`|`, `?`, `*`, and `+`), or a conditional choice between two protocol specifications (e `?` $s : s$), or parallel composition of protocol specifications (s `||` s and `!<`n`>` s)

The informal meaning of specifications is as follows: m denotes a call to m, w = m denotes a call to m where m's return value is stored in w, "s, s'" denotes the sequential composition of s and s', s | s' denotes s or s', s? denotes s zero or once, s* denotes s zero or many times, s+ denotes s one or many times, e ? s : s' denotes s if e is true s' otherwise, s || s' denotes s in parallel with s' (heterogeneous parallelism), and !<n> s denotes one to n s in parallel (homogeneous parallelism).

Since conditional protocols (case e ? s : s) are defined using expressions (which include specification values π i.e., logical variables α, see Sections 2.1 and 3.2.1), protocols are parameterized by class parameters. This allows to adapt a protocol to the different behaviors of a class (such as Section 3.6.2's `Iterator` which is parameterized by permission `p` to distinguish between read-only and read-write iterators). We use this feature in Section 7.3's first example.

For conciseness, we do not include a conditional without "else" branch even if we use it in later examples.

We extend the syntax of classes and interfaces with protocols:

$$
\begin{array}{rcll}
\mathit{prd} & ::= & \texttt{protocol}\ s\texttt{;} & \text{protocol definitions} \\
\mathit{cl} \in \mathsf{Class} & ::= & \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\ \ldots\ \mathit{prd}\ \ldots\ \texttt{\}} & \\
 & & & \text{(scope of } \bar{\alpha} \text{ includes } \bar{T}, U, \bar{V}, \mathit{prd}\text{)} \\
\mathit{int} \in \mathsf{Interface} & ::= & \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\ \ldots\ \mathit{prd}\ \ldots\ \texttt{\}} & \\
 & & & \text{(scope of } \bar{\alpha} \text{ includes } \bar{T}, \bar{U}, \mathit{prd}\text{)}
\end{array}
$$

To type check protocols, we extend the domain of type environments so that they contain protocol variables: From now on, type environments are partial functions of type $\mathsf{ObjId} \cup \mathsf{Var} \cup \mathsf{ProtVar} \rightharpoonup \mathsf{Type}$. The following rules type check protocols:

Well-typed Protocols, $\Gamma \vdash s : \diamond$

(Prot Mth)
$$\frac{\Gamma \vdash \texttt{this} : t\texttt{<}\bar{\alpha}\texttt{>} \quad \mathsf{mlkup}(m, t\texttt{<}\bar{\alpha}\texttt{>}) \neq \mathsf{undef}}{\Gamma \vdash m : \diamond}$$

(Prot Var Mth)
$$\frac{\Gamma \vdash \texttt{this} : t\texttt{<}\bar{\alpha}\texttt{>} \quad \Gamma \vdash w : U \quad \mathsf{mtype}(m, t\texttt{<}\bar{\alpha}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)}}{\Gamma \vdash w\ \texttt{=}\ m : \diamond}$$

(Prot Var Mth Infer)
$$\frac{\Gamma \vdash \texttt{this} : t\texttt{<}\bar{\alpha}\texttt{>} \quad \Gamma \vdash m : \diamond \quad w \notin \Gamma \quad \mathsf{mtype}(m, t\texttt{<}\bar{\alpha}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \mathit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} \quad \Gamma, w : U \vdash s : \diamond}{\Gamma \vdash w\ \texttt{=}\ m,\ s : \diamond}$$

(Prot Seq)
$$\frac{\Gamma \vdash s : \diamond \quad \Gamma \vdash s' : \diamond}{\Gamma \vdash s,\ s' : \diamond}$$

(Prot Or)
$$\frac{\Gamma \vdash s : \diamond \quad \Gamma \vdash s' : \diamond}{\Gamma \vdash s\ \texttt{|}\ s' : \diamond}$$

(Prot ?)
$$\frac{\Gamma \vdash s : \diamond}{\Gamma \vdash s\texttt{?} : \diamond}$$

(Prot *)
$$\frac{\Gamma \vdash s : \diamond}{\Gamma \vdash s\texttt{*} : \diamond}$$

(Prot +)
$$\frac{\Gamma \vdash s : \diamond}{\Gamma \vdash s\texttt{+} : \diamond}$$

(Prot If)
$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s : \diamond \quad \Gamma \vdash s' : \diamond}{\Gamma \vdash e\ \texttt{?}\ s : s' : \diamond}$$

(Prot ||)
$$\frac{\Gamma \vdash s : \diamond \quad \Gamma \vdash s' : \diamond}{\Gamma \vdash s\ \texttt{||}\ s' : \diamond}$$

$$\frac{\Gamma \vdash n : \texttt{int} \quad \Gamma \vdash s : \diamond}{\Gamma \vdash \texttt{!<}n\texttt{>}\ s : \diamond} \text{ (Prot !)}$$

Most rules are standard except (Prot Var Mth Infer). This rule is used to infer the type of protocol variables. To do this, the rule looks up the receiver's type ($t\texttt{<}\bar{\alpha}\texttt{>}$) and uses the receiver's type to look up the return type U of the method m (see the premise $\texttt{mtype}(m, t\texttt{<}\bar{\alpha}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \textit{spec}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)}$). The inferred type is then passed on to type check the continuation of the protocol ($\Gamma, w : U \vdash s' : \diamond$). Protocols are type checked when classes are verified. This is formally expressed by adding a premise to Section 3.4's (Cls) rule:

$$\frac{\ldots \quad \texttt{this} : C\texttt{<}\bar{\alpha}\texttt{>}, \bar{\alpha} : \bar{T} \vdash s : \diamond \quad \ldots}{\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\ \ldots\ \texttt{protocol}\ s\texttt{;}\ \ldots\ \texttt{\}} : \diamond} \text{ (Cls)}$$

7.3 Examples

In this section, we exemplify the use of protocols on two classes that we studied earlier in Chapter 3.

Iterator. First, we look at the `Iterator` interface from Section 3.6.2. For convenience, this interface is repeated here:

```
interface Iterator<perm p,Collection iteratee>{
 init (Collection c);
 boolean hasNext();
 Object next();
 void remove();
}
```

Clients of interface `Iterator` must follow a precise protocol. We already explained it informally at the level of method contracts in Section 3.6.2. In Java's documentation, however, the protocol is described in terms of method calls. This description is spread among the documentation of the methods of interface `Iterator` [80].

Intuitively, Java's documentation specifies that `init` must be called first, then `hasNext` must be called, then `next` may be called only if `hasNext` returned true before, and then `remove` may be called if the iterator is read-write (it can read and write to the iteratee), and this is to be repeated (except the call to `init`). To express that iterators may have write-access to the iteratee, we parameterized interface `Iterator` by a permission `p` (see details in Section 3.6.2). If `p` is instantiated by `1`, one obtains a read-write iterator, otherwise a read-only iterator.

We now show how to formally express this protocol with our extension. The fact that `init` must be called first, before `hasNext` is expressed by this protocol:

```
protocol init, hasNext;
```

Then, the fact that `next` should be called if `hasNext` returned true before is described by this protocol:

```
protocol init, w = hasNext, w ? next;
```

Above, we use protocol variable `w` to bind the value returned by the call to `hasNext`. Next, we express that `remove` may be called after next if the iterator is read-write with the following protocol:

```
protocol init, w = hasNext, w ? (next, p==1 ? (remove?));
```

To complete the protocol, we specify that it can be repeated 0 or more times (excluding `init`):

```
protocol init, (w = hasNext, w ? (next, p==1? (remove?)))*;
```

We believe that such a formal specification (compared to Java's informal documentation) would help clients to use `Iterator`s in a disciplined way.

`Roster`. Our second example is the `Roster` interface from Section 3.6.1 that collects student identifiers and associates. For convenience, we repeat this interface:

```
interface Roster{
 void updateGrade(int id, int grade);
 boolean contains(int id);
}
```

For performance issues, implementers of the `Roster` interface should allow multiple threads to concurrently read a roster and a thread to update the grades while other threads concurrently read the student identifiers. This can be specified with the following protocol:

```
protocol (updateGrade?  || (!contains)?)*;
```

Above, we abusively write `!contains` for `!<2^32>contains`. Our choice is motivated by Java's specification: the maximal number of threads for most virtual machines implementation is theoretically 2^{32}. Hence, our desugaring for the unparameterized `!` operator is sound for such virtual machines. In practice, virtual machines cannot handle such a big number of threads and would crash before this number of threads.

Intuitively, in terms of method contracts, interface `Roster`'s protocol constrains method `updateGrade` and method `contains` not to interfere. In terms of predicates, it requires that the state obtained after creating a `Roster` instance can be divided into (1) the state required by the execution of `updateGrade` and (2) the state required by multiple concurrent executions of `contains`.

7.4 Semantics

Now, we give the semantics of protocols. This semantics provides a way to check program-generated traces against protocols.

Our semantics consists of two different parts: (1) in Section 7.4.1, we instrument programs to generate traces that track sequences of methods calls on all objects and (2) in Section 7.4.2, we define the semantics of protocols in terms of predicates on traces. Finally, in Section 7.4.3, we relate protocols and programs: we define what it means for the run of a program to respect a protocol.

7.4.1 A Trace Semantics for Multithreaded Programs

To generate traces, we modify Section 5.2's operational semantics to keep track of sequences of method calls on objects.

Runtime Structures. Our semantics is defined in terms of *actions*, *ghost stores*, and *traces*. An action is either method entry or method exit. Ghost stores keep track of class parameters and protocol variables. Class parameters are final: they are assigned only once (at object creation), while protocol variables are assigned each time the corresponding method is called. As an example, in interface `Iterator`'s protocol, the protocol variable `w` is assigned each time `hasNext` is called. Ghost stores map logical variables (generic parameters) and protocol variables to specification values (permissions, client-defined values, and Java built-in values). Finally, a trace is a sequence of ghost stores and actions:

$$\begin{array}{rcl} a \in \mathsf{Action} & ::= & m.\texttt{enter} \mid m.\texttt{exit} \\ \sigma \in \mathsf{GhostStore} & ::= & (\mathsf{LogVar} \cup \mathsf{ProtVar}) \rightharpoonup \mathsf{SpecVal} \\ \tau \in \mathsf{Trace} & ::= & \overline{\mathsf{GhostStore} \times \mathsf{Action}} \end{array}$$

Alternatively, we could use heaps in our runtime structures, because they already keep track of class parameters. We choose this solution, however, because we have to keep track of protocol variables, which are not included in heaps anyway.

Pairs of a ghost store σ[1] and an action a are written (σ, a). We write $(\sigma, a) \cdot \tau$ for the concatenation of the pair (σ, a) and trace τ, we abusively write $\tau \cdot \tau'$ for the concatenation of trace τ and trace τ' (this can be trivially defined in terms of the constructor $(\sigma, a) \cdot \tau$), and we write ϵ for the empty sequence or trace. Note that, given the execution of a multithreaded program and a distinguished object, we do not differentiate separate actions from different threads. All actions of all threads form a single trace. This suffices to express the semantics of protocols.

In the operational semantics, we associate each object with a trace. For this, we use trace tables $tt \in \mathsf{TraceTable} = \mathsf{ObjId} \rightharpoonup \mathsf{Trace}$. We extend Section 5.2's states to include a trace table:

$$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{LockTable} \times \mathsf{TraceTable} \times \mathsf{ThreadPool}$$

We need to modify Section 5.2's rule for method call (Red Call) to keep track of method calls. We have to be careful though: we do not want to keep track of all method calls, we only want to keep track of clients's method calls i.e., top level calls. For example, if a client calls $o.m()$ and m's implementation reentrantly calls $o.n()$, we only keep track of the call to m (we check protocols w.r.t. to client calls, not internal calls). To distinguish between client calls and internal (i.e., reentrant) calls, we use *closed* traces. A closed trace is a sequence of pairs of an action and a ghost store such that any $m.\texttt{enter}$ action is followed by a $m.\texttt{exit}$ action.

We write $\mathsf{closed}(\tau)$ to denote closed traces. Formally, closed is defined as follows:

Closed Traces, $\mathsf{closed}(\tau)$**:**

$$\frac{}{\mathsf{closed}(\epsilon)}\ (\text{Closed } \epsilon) \qquad \frac{}{\mathsf{closed}((\sigma, m.\texttt{enter}) \cdot (\sigma', m.\texttt{exit}))}\ (\text{Pair Closed}) \qquad \frac{\mathsf{closed}(\tau) \quad \mathsf{closed}(\tau')}{\mathsf{closed}(\tau \cdot \tau')}\ (\text{Conc Closed})$$

[1] We overload σ, which we previously used to denote substitutions.

In the operational semantics below, we use the following functions:

$$(\sigma, a)_1 \triangleq \sigma \qquad (\sigma, a)_2 \triangleq a$$

$$\begin{array}{rcl} \mathsf{last}(\epsilon) & = & \mathsf{undef} \\ \mathsf{last}((\sigma, a) \cdot \tau) & \triangleq & \begin{cases} \mathsf{last}(\tau) & \text{iff } \tau \neq \epsilon \\ (\sigma, a) & \text{otherwise} \end{cases} \end{array}$$

When we extend object o's trace in a trace table tt, we use the following abbreviation:

$$tt[o \leftarrow (\sigma, a)] \triangleq tt[o \mapsto tt(o) \cdot (\sigma, a)]$$

Initialization. We modify Section 5.2's definition of the initial state of a program. Initially, the trace table is empty (hence the third $\emptyset$):

$$\mathsf{init}(c) = \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\},\ \emptyset,\ \emptyset,\ \mathsf{main\ is}\ (\emptyset\ \mathsf{in}\ c) \rangle$$

Operational Semantics. We modify Section 5.2's semantics to generate traces. Recall that, a long time ago in a section far far away (i.e., Section 2.2); we defined a derived form $\ell \leftarrow c;\ c'$, which assigns the result of a computation c to variable ℓ. In the operational semantics, this derived form is introduced when a method (with body c) call is executed (see (Red Call) on page 70).

We define a new derived form that has the same purpose as $\leftarrow$. This new derived form, however, also indicates that the method call being performed is a client call. This is needed because, when a method returns, we sometimes need to keep track of the value returned (because of possible protocol variables). To model this, we introduce a new derived form $\ell \twoheadleftarrow_o c;\ c'$ to mean that (1) c is the body of the method about to execute; (2) when the method will return, we will have to keep track of its return value; and (3) the method's receiver is o. Derived form $\twoheadleftarrow$ is defined similarly to $\leftarrow$ except in the base case $\ell \twoheadleftarrow_o v;\ c$:

$$\begin{array}{rcll} \ell \twoheadleftarrow_o v;\ c & \triangleq & \ell = \texttt{return-and-store}_o(v);\ c & \\ \ell \twoheadleftarrow_o (T\ \ell';\ c);\ c' & \triangleq & T\ \ell';\ \ell \twoheadleftarrow_o c;\ c' & \text{if } \ell' \notin \mathsf{fv}(c'),\ \ell' \neq \ell \\ \ell \twoheadleftarrow_o (T\ \imath = \ell';\ c);\ c' & \triangleq & T\ \imath = \ell';\ \ell \twoheadleftarrow_o c;\ c' & \text{if } \imath \notin \mathsf{fv}(c') \\ \ell \twoheadleftarrow_o (hc;\ c);\ c' & \triangleq & hc;\ \ell \twoheadleftarrow_o c;\ c' & \end{array}$$

Above, `return-and-store` is a new command. Like `return` (which was introduced in Section 2.2 on page 12), `return-and-store` is used to indicate when the receiver changes. In addition, `return-and-store` indicates that (1) the value being returned must be stored in the trace table and (2) the receiver of the returning method was o:

c ::= ... | $\ell = \texttt{return-and-store}_o(v);\ c$ | ...
Restriction: This clause must not occur in source programs.

In the *operational semantics*, we change rule (Red New), we replace rule (Red Call) by rules (Red Call Track) and (Red Call Do Not Track), and we add rule (Red Return Track). Rule (Red Return Track) uses function protlkup to lookup protocols in the class table. Function protlkup is formally defined in Appendix A.

State Reductions, $st \rightarrow_{ct} st'$**:**

...

(Red New) $o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{initStore}(C\texttt{<}\bar{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o]$
$l' = l[o \mapsto \mathsf{free}] \quad \sigma = \bar{\alpha} \mapsto \bar{\pi} \quad \tau = (\sigma, \texttt{new.enter}) \cdot (\sigma, \texttt{new.exit})$

$$\langle h,\ l,\ tt,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = \texttt{new}\ C\texttt{<}\bar{\pi}\texttt{>};\ c)\rangle \rightarrow \langle h',\ l',\ tt[o \mapsto \tau],\ ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$$

(Red Call Track) $m \notin \{\texttt{fork}, \texttt{wait}, \texttt{notify}\} \quad h(o)_1 = C\texttt{<}\bar{\pi}'\texttt{>}$
$\mathsf{mbody}(m, C\texttt{<}\bar{\pi}'\texttt{>}) = (\imath_0, \bar{\imath}).c_m \quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}]$
$\mathsf{closed}(tt(o)) \quad tt' = tt[o \leftarrow (\mathsf{last}(tt(o))_1, m.\texttt{enter})]$

$$\langle h,\ l,\ tt,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.m(\bar{v});\ c)\rangle \rightarrow \langle h,\ l,\ tt',\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \twoheadleftarrow_o c';\ c)\rangle$$

(Red Call Do Not Track) $m \notin \{\texttt{fork}, \texttt{wait}, \texttt{notify}\} \quad h(o)_1 = C\texttt{<}\bar{\pi}'\texttt{>}$
$\mathsf{mbody}(m, C\texttt{<}\bar{\pi}'\texttt{>}) = (\imath_0, \bar{\imath}).c_m \quad c' = c_m[o/\imath_0, \bar{v}/\bar{\imath}] \quad \neg\mathsf{closed}(tt(o))$

$$\langle h,\ l,\ tt,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = o.m(\bar{v});\ c)\rangle \rightarrow \langle h,\ l,\ tt,\ ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \leftarrow c';\ c)\rangle$$

(Red Return Track) $h(o)_1 = C\texttt{<}\bar{\pi}\texttt{>} \quad w_0 = m, \ldots, w_n = m \in \mathsf{protlkup}(C\texttt{<}\bar{\pi}\texttt{>})$
$\mathsf{last}(tt(o)) = (\sigma, m.\texttt{enter}) \quad tt' = tt[o \leftarrow (\sigma[w_0 \mapsto v, \ldots, w_n \mapsto v], m.\texttt{exit})]$

$$\langle h,\ l,\ tt,\ p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = \texttt{return-and-store}_o(v);\ c)\rangle \rightarrow \langle h,\ l,\ tt',\ p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell = v;\ c)\rangle$$

...

Remarks.

- Even though `new` is not a method call, the trace of an object is initialized as soon as it is created. The initial trace of an object consists of two pairs of a ghost store and an action. We need to do so because we need to keep track of the parameters used to instantiate the newly created object (which are stored in the ghost store σ). The two actions are simply `new.enter` and `new.exit`.
- When calling a method, if $\mathsf{closed}(tt(o))$ holds, the method call is tracked because it is a client call: rule (Red Call Track) applies. In this case, the trace table is updated at o: a new pair of a ghost store (the last ghost store of o's trace $tt(o)$ i.e., $\mathsf{last}(tt(o))_1$) and an action is concatenated to o's trace. Derived form $\twoheadleftarrow$ is used to indicate that the method call is tracked.
- When calling a method, iff $\mathsf{closed}(tt(o))$ does not hold, the method call is not tracked because it is not a client call: rule (Red Call Do Not Track) applies. Derived form $\leftarrow$ is used to indicate that the method call is not tracked.
- When a method whose return value must be tracked terminates, rule (Red Return Track) applies. In this rule, the set of protocol variables $(w_0, \ldots, w_n)$ that must be updated is looked up by $\mathsf{protlkup}(C\texttt{<}\bar{\pi}\texttt{>})$. Type $C\texttt{<}\bar{\pi}\texttt{>}$ is the type of the receiver (i.e., o) of the method returning (as indicated by subscript o on `return-and-store`).

Example. We show how the operational semantics behaves w.r.t. to traces. Our example reuses the `Iterator` protocol explained in Section 7.3. For simplicity, in states we focus on trace tables. We assume that the trace table is initially empty. Further, we suppose that collection `c` has address a_c in the heap, that the newly created iterator is

allocated at address o, and that `i.hasNext()` returns `true`. Finally, we use the following abbreviations:

$$\begin{array}{lcl}
\sigma & \triangleq & o \mapsto (\mathtt{p} \mapsto 1/2, \mathtt{c} \mapsto a_c) \\
\sigma' & \triangleq & o \mapsto (\mathtt{p} \mapsto 1/2, \mathtt{c} \mapsto a_c, \mathtt{w} \mapsto \mathtt{true}) \\
 & & \\
\tau_{\mathtt{new}} & \triangleq & (\sigma, \mathtt{new.enter}) \cdot (\sigma, \mathtt{new.exit}) \\
\tau_{\mathtt{init}} & \triangleq & (\sigma, \mathtt{init.enter}) \cdot (\sigma, \mathtt{init.exit}) \\
\tau_{\mathtt{hasNext}} & \triangleq & (\sigma, \mathtt{hasNext.enter}) \cdot (\sigma', \mathtt{hasNext.exit}) \\
\tau_{\mathtt{next}} & \triangleq & (\sigma', \mathtt{next.enter}) \cdot (\sigma', \mathtt{next.exit})
\end{array}$$

Below, the left hand side shows the code while the right hand side shows the state in between consecutive commands.

```
void getFirst(Collection c){
  Iterator i = new Iterator<1/2,c>;
  i.init(c);
  bool b = i.hasNext();
  Object o = i.next();
}
```

$\leftarrow \langle \emptyset \rangle$
$\leftarrow \langle \tau_{\mathtt{new}} \rangle$
$\leftarrow \langle \tau_{\mathtt{new}} \cdot \tau_{\mathtt{init}} \rangle$
$\leftarrow \langle \tau_{\mathtt{new}} \cdot \tau_{\mathtt{init}} \cdot \tau_{\mathtt{hasNext}} \rangle$
$\leftarrow \langle \tau_{\mathtt{new}} \cdot \tau_{\mathtt{init}} \cdot \tau_{\mathtt{hasNext}} \cdot \tau_{\mathtt{next}} \rangle$

Operational Semantics. In addition to user-defined calls, we have to keep track of "special calls" i.e., `join`, `lock`, `unlock`, `wait`, and `notify`. Methods `join`, `wait`, and `notify` are special, because the operational semantics treats them in a special way. Methods `lock` and `unlock` are also special: they are formally defined as commands (i.e., they are not in class `Object`) and the operational semantics treats them in a special way.

We could simply forbid protocols to mention these "special" methods. They do not, however, raise any problem for our technique. Further, we believe that programmers could find it useful to mention these special methods in protocols. For example, if a programmer wants to have non-reentrant locks, he could use the following class (recall that methods `lock` and `unlock` are available in all classes):

```
class NonReentrantLock{
 protocol (lock, unlock)*;
}
```

Integrating the aforementioned methods into our framework for protocols requires crafting new rules for the operational semantics. We do not show these rules which would be similar to the operational semantics's rules for the methods considered (see (Red Join) on page 51; (Red Lock), (Red Unlock), (Red Wait), (Red Notify), and (Red Skip Notify) on page 70). The only change to these rules would be to keep track of calls to these methods if the trace of the receiver object is closed.

Checking that Program Respect Protocols. So far, we have given a semantics of protocols in terms of program runs; but we did not give any algorithm to actually check that programs respect protocols. Cheon and Perumandla [32] give an algorithm to do that by adding extra runtime checks. We could adapt this technique in a straightforward way. The only difference is that we would need to synchronize access to the data structures used by the runtime checker.

7.4.2 Semantics of Protocols

The semantics of protocols is given by $[\![.]\!] : \mathsf{ProtSpec} \to \overline{\mathsf{GhostStore}} \to 2^{\overline{\mathsf{Action}}}$. Intuitively, $[\![s]\!](\overline{\sigma})$ returns the set of all possible sequences of actions that satisfy s w.r.t. $\overline{\sigma}$. Except for the cases e `?` $s : s$, s `||` s, and `!<`n`>` s, $[\![.]\!]$'s definition is standard:

$$
\begin{array}{rcl}
[\![m]\!](\overline{\sigma}) & \triangleq & \begin{cases} \{m.\texttt{enter} \cdot m.\texttt{exit}\} & \text{iff } \overline{\sigma} = \sigma' \cdot \sigma'' \\ \mathsf{undef} & \text{otherwise} \end{cases} \\
[\![w \texttt{ = } m]\!](\overline{\sigma}) & \triangleq & \begin{cases} \{m.\texttt{enter} \cdot m.\texttt{exit}\} & \text{iff } \overline{\sigma} = \sigma' \cdot \sigma'' \\ \mathsf{undef} & \text{otherwise} \end{cases} \\
[\![s,\ s']\!](\overline{\sigma}) & \triangleq & \left\{ \overline{a} \cdot \overline{a'} \;\middle|\; \begin{array}{l} \overline{a} \in [\![s]\!](\overline{\sigma_0}) \\ \overline{a'} \in [\![s']\!](\overline{\sigma_1}) \\ \overline{\sigma_0} \cdot \overline{\sigma_1} = \overline{\sigma} \end{array} \right\} \\
[\![s \texttt{ | } s']\!](\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \cup [\![s']\!](\overline{\sigma}) \\
[\![s\texttt{?}]\!](\overline{\sigma}) & \triangleq & \{\epsilon\} \cup [\![s]\!](\overline{\sigma}) \\
[\![s\texttt{*}]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \mathbb{N}} [\![s]\!]^i(\overline{\sigma}) \\
[\![s\texttt{+}]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \mathbb{N}^+} [\![s]\!]^i(\overline{\sigma})
\end{array}
$$

and $[\![s]\!]^n(\overline{\sigma})$ is defined as follows:

$$
\begin{array}{rcl}
[\![s]\!]^0(\epsilon) & \triangleq & \{\epsilon\} \\
[\![s]\!]^i(\overline{\sigma}) & \triangleq & \left\{ \overline{a} \cdot \overline{a'} \;\middle|\; \begin{array}{l} \overline{a} \in [\![s]\!](\overline{\sigma_0}) \\ \overline{a'} \in [\![s]\!]^{i-1}(\overline{\sigma_1}) \\ \overline{\sigma_0} \cdot \overline{\sigma_1} = \overline{\sigma} \end{array} \right\}
\end{array}
$$

To define the semantics of a conditional protocol, we define function $\mathsf{first} : \overline{\mathsf{GhostStore}} \rightharpoonup \mathsf{GhostStore}$ to look up the first element of a sequence of ghost stores:

$$\mathsf{first}(\epsilon) \quad = \quad \mathsf{undef} \qquad \mathsf{first}(\sigma \cdot \overline{\sigma}) \quad \triangleq \quad \sigma$$

In the semantics of conditional protocols, we check the conditional's truth w.r.t. the first ghost store of the sequence of ghost stores and choose the protocol's continuation accordingly:

$$
[\![e \texttt{ ? } s : s']\!](\overline{\sigma}) \quad \triangleq \quad \begin{cases} [\![s]\!](\overline{\sigma}) & \text{iff} \quad [\![e]\!](\mathsf{first}(\overline{\sigma})) = \texttt{true} \\ [\![s']\!](\overline{\sigma}) & \text{otherwise} \end{cases}
$$

To define the cases s `||` s and `!<`n`>` s of the semantics of specifications, we define the interleaving of two sequences of actions with $\between : \overline{\mathsf{Action}} \times \overline{\mathsf{Action}} \to 2^{\overline{\mathsf{Action}}}$:

$$
\begin{array}{rcl}
\epsilon \between a & \triangleq & \{a\} \\
a \between \epsilon & \triangleq & \{a\} \\
a \cdot \overline{a} \between a' \cdot \overline{a'} & \triangleq & \begin{array}{c} \{a \cdot \overline{a''} \mid \overline{a''} \in \overline{a} \between a' \cdot \overline{a'}\} \\ \cup \\ \{a' \cdot \overline{a''} \mid \overline{a''} \in a \cdot \overline{a} \between \overline{a'}\} \end{array}
\end{array}
$$

The $\between$ operator is extended to sets of sequences ($\between : 2^{\overline{\mathsf{Action}}} \times 2^{\overline{\mathsf{Action}}} \to 2^{\overline{\mathsf{Action}}}$) in the straightforward way. Then, we can define:

$$
\begin{array}{rcl}
[\![s \texttt{ || } s']\!](\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \between [\![s']\!](\overline{\sigma}) \\
[\![\texttt{!<}n\texttt{>}\ s]\!](\overline{\sigma}) & \triangleq & \bigcup_{i \in \{1,2,\ldots,n\}} \between^i s(\overline{\sigma})
\end{array}
$$

where $\text{M}\backslash^i : \mathsf{ProtSpec} \to 2^{\overline{\mathsf{Action}}}$ is defined as follows:

$$\begin{array}{lcl} \text{M}\backslash^1\ s(\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma}) \\ \text{M}\backslash^i\ s(\overline{\sigma}) & \triangleq & [\![s]\!](\overline{\sigma})\ \text{M}\backslash\ (\text{M}\backslash^{i-1}\ s(\overline{\sigma})) \quad (\text{for } i \geq 2) \end{array}$$

7.4.3 Satisfaction of Traces w.r.t. Protocols

A trace satisfies a protocol if its underlying sequence of actions is the prefix of one of the sequences of the protocol's semantics. We use the *prefix* of one of the sequences of the protocol's semantics, because we consider that not terminating a protocol is harmless. Formally:

$$(\sigma_0, a_0) \ldots (\sigma_n, a_n) \vdash s \quad \text{iff} \quad (\exists\, \overline{a}\ \overline{\sigma})(a_0 \ldots a_n \cdot \overline{a} \in [\![\texttt{new},\ \texttt{s}]\!](\sigma_0 \ldots \sigma_\texttt{n} \cdot \overline{\sigma}))$$

In our definition of $\vdash$, we add method `new` as a prefix of the protocol being checked. We need to do that because as rule (Red New) on page 124 shows, when an object is created, its initial trace is initialized with actions `new.enter` and `new.exit`.

Example. We show how the trace generated by method `getFirst` from Section 7.4.1 satisfies interface `Iterator`'s protocol. For convenience, we repeat some abbreviations we used to define this trace:

$$\begin{array}{lcl} \sigma & \triangleq & o \mapsto (\texttt{p} \mapsto 1/2, \texttt{c} \mapsto a_c) \\ \sigma' & \triangleq & o \mapsto (\texttt{p} \mapsto 1/2, \texttt{c} \mapsto a_c, \texttt{w} \mapsto \texttt{true}) \\ \\ \tau_{\texttt{new}} & \triangleq & (\sigma, \texttt{new.enter}) \cdot (\sigma, \texttt{new.exit}) \\ \tau_{\texttt{init}} & \triangleq & (\sigma, \texttt{init.enter}) \cdot (\sigma, \texttt{init.exit}) \\ \tau_{\texttt{hasNext}} & \triangleq & (\sigma, \texttt{hasNext.enter}) \cdot (\sigma', \texttt{hasNext.exit}) \\ \tau_{\texttt{next}} & \triangleq & (\sigma', \texttt{next.enter}) \cdot (\sigma', \texttt{next.exit}) \end{array}$$

We have to show:

$$\tau_{\texttt{new}} \cdot \tau_{\texttt{init}} \cdot \tau_{\texttt{hasNext}} \cdot \tau_{\texttt{next}} \vdash \texttt{init, (w = hasNext, w ? (next, p==1? (remove?)))*}$$

We introduce the following abbreviations:

$$\begin{array}{lcl} \overline{a_0} & \triangleq & \texttt{new.enter} \cdot \texttt{new.exit} \cdot \texttt{init.enter} \cdot \texttt{init.exit} \\ \overline{a_3} & \triangleq & a_0 \cdot \texttt{hasNext.enter} \cdot \texttt{hasNext.exit} \cdot \texttt{next.enter} \cdot \texttt{next.exit} \\ \\ s_0 & \triangleq & \texttt{new, init, (w = hasNext, w ? (next, p==1? (remove?)))*} \\ s_1 & \triangleq & \texttt{init, (w = hasNext, w ? (next, p==1? (remove?)))*} \\ s_2 & \triangleq & \texttt{(w = hasNext, w ? (next, p==1? (remove?)))*} \\ s_3 & \triangleq & \texttt{w = hasNext, w ? (next, p==1? (remove?))} \end{array}$$

By definition of $\vdash$, we have to find $\overline{a}^c$, $\overline{\sigma}^c$ such that the following statements hold (read "c" as "continuation"):

(b) $a_3 \cdot \overline{a_c} \in [\![s_0]\!](\sigma \cdot \sigma \cdot \sigma \cdot \sigma \cdot \sigma \cdot \sigma' \cdot \sigma' \cdot \sigma' \cdot \overline{\sigma_c})$

We choose $\overline{a}^c = \overline{\sigma}^c = \epsilon$. By case "," of our semantics, we have to find $\overline{a}_0^?, \overline{\sigma}_0^?, \overline{a}_1^?$, and $\overline{\sigma}_1^?$ such that the following statements hold:

(c) $\overline{a}_0^? \in [\![\texttt{new}]\!](\overline{\sigma}_0^?)$
(d) $\overline{a}_1^? \in [\![s_1]\!](\overline{\sigma}_1^?)$
(e) $\overline{a}_0^? \cdot \overline{a}_1^? = \overline{a}_3$ and $\overline{\sigma}_0^? \cdot \overline{\sigma}_1^? = \sigma \cdot \sigma \cdot \sigma \cdot \sigma \cdot \sigma \cdot \sigma \cdot \sigma' \cdot \sigma' \cdot \sigma'$.

We choose:

- $\overline{a}_0^? = \texttt{new.enter} \cdot \texttt{new.exit}$
- $\overline{\sigma}_0^? = \sigma \cdot \sigma$
- $\overline{a}_1^? = \texttt{init.enter} \cdot \texttt{init.exit} \cdot \texttt{hasNext.enter} \cdot \texttt{hasNext.exit} \cdot \texttt{next.enter} \cdot \texttt{next.exit}$
- $\overline{\sigma}_1^? = \sigma \cdot \sigma \cdot \sigma \cdot \sigma' \cdot \sigma' \cdot \sigma'$

By case method call, goal (c) is closed. Further, goal (e) is trivially closed. To show goal (d), we remark that the following statements hold:

(f) $\texttt{init.enter} \cdot \texttt{init.exit} \in [\![\texttt{init}]\!](\sigma \cdot \sigma)$
(g) $\texttt{hasNext.enter} \cdot \texttt{hasNext.exit} \cdot \texttt{next.enter} \cdot \texttt{next.exit} \in [\![s_2]\!](\sigma \cdot \sigma' \cdot \sigma' \cdot \sigma')$

Goal (f) holds by the case for method call of our semantics. We are left with goal (g). By case "*" of our semantics, goal (g) follows from the following statements:

(h) $\bigcup_{i \in [0,1]} [\![s_3]\!]^i(\sigma \cdot \sigma' \cdot \sigma' \cdot \sigma') \in [\![(s_2)*]\!](\sigma \cdot \sigma' \cdot \sigma' \cdot \sigma')$
(i) $\texttt{hasNext.enter} \cdot \texttt{hasNext.exit} \cdot \texttt{next.enter} \cdot \texttt{next.exit} \in \bigcup_{\texttt{i} \in [0,1]} [\![s_3]\!]^{\texttt{i}}(\sigma \cdot \sigma' \cdot \sigma' \cdot \sigma')$

By case "," of our semantics, goal (i) follows from the following statements:

(j) $\texttt{hasNext.enter} \cdot \texttt{hasNext.exit} \in [\![\texttt{w} = \texttt{hasNext}]\!](\sigma \cdot \sigma')$
(k) $\texttt{next.enter} \cdot \texttt{next.exit} \in [\![\texttt{w ? (next, p==1? (remove?))}]\!](\sigma' \cdot \sigma')$

Goal (j) holds by the case for method call of our semantics. By the case for conditional protocols of our semantics (case e `?` s : s'), to prove that goal (g) holds, it suffices to prove that the following statements hold:

(l) $[\![\texttt{w}]\!](\mathsf{first}(\sigma' \cdot \sigma')) = \texttt{true}$
(m) $\texttt{next.enter} \cdot \texttt{next.exit} \in [\![\texttt{next, p==1? (remove?)}]\!](\sigma' \cdot \sigma')$

Goal (l) holds because, by definition of σ', we have: $[\![\texttt{w}]\!](\mathsf{first}(\sigma' \cdot \sigma')) = [\![\texttt{w}]\!](\sigma') = \sigma'(\texttt{w}) = \texttt{true}$. Goal (m) holds because the following statements hold:

(n) $\texttt{next.enter} \cdot \texttt{next.exit} \in [\![\texttt{next}]\!](\sigma' \cdot \sigma')$
(o) $\epsilon \in [\![\texttt{p==1? (remove?)}]\!](\epsilon)$

Goal (n) holds by the case for method calls of our semantics. Finally, goal (o) holds by the case for conditional protocols and the case `?` of our semantics.

Subtyping. We did not mention subtyping earlier because it is unproblematic. As in the previous work on method call sequences [32], a straightforward interpretation of protocols inheritance is to conjoin inherited protocols to the protocols declared in the inheriting class. This means that a class has to respect inherited protocols and has to respect its own protocols.

7.5 Checking Method Contracts against Protocols

To help programmers check that method contracts are correct w.r.t. a protocol, we generate a program that simulates the protocol. The generated program is like a "maximal" program, because it simulates all programs that obey the protocol. If the generated program cannot be proven correct, the method contracts are wrong: some (client-provided) programs will fail to verify, even though they obey the protocol considered.

When generating programs, we cannot (easily) provide method parameters fulfilling the part of method preconditions that is relevant to method parameters. Therefore, our technique is restricted to methods whose precondition has the form $F\ op\ G$ (where $op = \{\texttt{*}, \texttt{\&}\}$) and no method parameter (excluding `this`) occurs in F. We refer to this restriction by saying that method preconditions must be *receiver splittable*. Because of this restriction, we can syntactically split method contracts into a part that concerns the receiver (F) and a part that concerns the parameters (G). Then, when checking that method contracts are correct w.r.t. protocols, the part of the preconditions relevant to parameters (G) is dropped. We believe that, in practice, most method preconditions are *receiver splittable*. Our belief is supported by the fact that all examples in this thesis are *receiver splittable*. In addition, all examples from Parkinson's thesis [88] are *receiver splittable*.

We write $\mathsf{recs}(F)$ in $t\texttt{<}\bar{\alpha}\texttt{>}.m$ to denote that precondition F of method m in class (or interface) $t\texttt{<}\bar{\alpha}\texttt{>}$ is receiver splittable. Formally, judgment recs is defined as follows:

Receiver-Splittable Formulas, $\mathsf{recs}(F)$ in $t\texttt{<}\bar{\alpha}\texttt{>}.m$

(Recs Com)

$$\frac{op \in \{\texttt{*}, \texttt{\&}\} \quad \mathsf{recs}(G\ op\ F) \text{ in } t\texttt{<}\bar{\alpha}\texttt{>}.m}{\mathsf{recs}(F\ op\ G) \text{ in } t\texttt{<}\bar{\alpha}\texttt{>}.m}$$

(Recs Base)

$$\frac{op \in \{\texttt{*}, \texttt{\&}\} \quad \mathsf{mtype}(m, t\texttt{<}\bar{\alpha}\texttt{>}) \triangleq \texttt{<}\bar{T}\ \bar{\alpha}'\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} \quad \texttt{this} \notin G \quad \mathsf{fv}(F) \subseteq \{\texttt{this}, \bar{\alpha}, \bar{\alpha}'\}}{\mathsf{recs}(F\ op\ G) \text{ in } t\texttt{<}\bar{\alpha}\texttt{>}.m}$$

Rule (Recs Base) above formalizes our notion of receiver-splittable. In this rule, F indicates the part of $F\ op\ G$ that concerns the receiver; while G indicates the part that concerns method parameters.

For our purposes, considering only the part of method preconditions that is relevant to the receiver is sound: intuitively, that is because, when verifying generated programs, verification is easier than in the real world (preconditions are weakened). If a generated program fails to verify, the corresponding real world program (i.e., where preconditions are complete) will also fail to verify.

We now have all the machinery to generate programs that have to be verified to check that method contracts are correct w.r.t. protocols.

Figure 7.1 on page 130 shows the rules for generating programs. Function $\mathsf{gen}(r, s, T)$ generates the program to be proven correct to show that method contracts of class T are correct w.r.t. protocol s. In $\mathsf{gen}(r, s, T)$, r denotes the receiver. In the cases m and w = m, "unknown parameters" (parameters about which nothing is known) are passed to m. Such parameters are modeled by using an alternative rule for variable declaration:

$\mathsf{gen}(r, \texttt{new}, T) \triangleq T_0\ \alpha_0;\ \ldots;\ T_n\ \alpha_n;\ r\texttt{=new<}\alpha_0, \ldots, \alpha_n\texttt{>}t;$

$\mathsf{gen}(r, m, T) \triangleq T_0\ \ell_0;\ \ldots;\ T_n\ \ell_n;\ T''o\ \texttt{=}\ r.m(\ell_0, \ldots, \ell_n);$ $(m \neq \texttt{new})$

$\mathsf{gen}(r, w\ \texttt{=}\ m, T) \triangleq T_0\ \ell_0;\ \ldots;\ T_n\ \ell_n;\ r.w\ \texttt{=}\ m(\ell_0, \ldots, \ell_n);\ T''o\ \texttt{=}\ r.w;$

$\mathsf{gen}(r, s,\ s', T) \triangleq \mathsf{gen}(r, s, T);\ \mathsf{gen}(r, s', T)$

$\mathsf{gen}(r, s\ \texttt{|}\ s', T) \triangleq \texttt{boolean b; if(b)\{}\ \mathsf{gen}(r, s, T)\ \texttt{\}else\{}\ \mathsf{gen}(r, s', T)\ \texttt{\}}$

$\mathsf{gen}(r, s\texttt{?}, T) \triangleq \texttt{boolean b; if(b)\{}\ \mathsf{gen}(r, s, T)\ \texttt{\}}$

$\mathsf{gen}(r, s\texttt{*}, T) \triangleq \texttt{boolean b; while(b)\{}\ \mathsf{gen}(r, s, T)\ \texttt{\}}$

$\mathsf{gen}(r, s\texttt{+}, T) \triangleq \mathsf{gen}(r, s, T);\ \texttt{boolean b; while(b)\{}\ \mathsf{gen}(r, s, T)\ \texttt{\}}$

$\mathsf{gen}(r, e\ \texttt{?}\ s : s', T) \triangleq$
```
boolean b;
if(b){ assume(e[r/this]); gen(r, s, T) }
else{ assume(!e[r/this]); gen(r, s', T) }
```

$\mathsf{gen}(r, s\ \texttt{||}\ s', T) \triangleq$
```
ThreadS tS=new ThreadS; ThreadSp tSp=new ThreadSp;
tS.init(r); tSp.init(r);
tS.start(); tSp.start();
tS.join(); tSp.join()
```
Above, class `ThreadS` is $\mathsf{classgen}(\texttt{ThreadS}, s, T)$ and Class `ThreadSp` is $\mathsf{classgen}(\texttt{ThreadSp}, s', T)$

$\mathsf{gen}(r, \texttt{!<}n\texttt{>}\ s, T) \triangleq$
```
ThreadS i_1= newThreadS; ...; ThreadS i_n= newThreadS;
i_1.init(r); ...; i_n.init(r);
i_1.start(); ...; i_n.start();
i_1.join(); ...; i_n.join();
```
Above, class `ThreadS` is $\mathsf{classgen}(\texttt{ThreadS}, s, T)$

$\mathsf{classgen}(C, s, T) \triangleq$
```
class C extends Thread{
 T rec;
 requires ?; ensures ?; void init(T v){ rec=v; }
 requires ?; ensures ?; void run(){ gen(rec, s, T) }
}
```

Conventions:
(1) In the first case, $T = t\texttt{<}\bar{\alpha}\texttt{>}$ and $\alpha_0, \ldots, \alpha_n = \bar{\alpha}$.
(2) In the second and third cases, T'' is m's return type and $T_0, \ldots, T_n$ are m's arguments types.
(3) In case $\texttt{!<}n\texttt{>}\ s$, $i_1 = 1, \ldots, i_n = n$
(4) All introduced names are fresh.

Figure 7.1: Program generation to check adherence of contracts to protocol s of receiver r of class T

$$\frac{\ell \notin F, G \quad \Gamma, \ell : T; v \vdash \{F\}c : U\{G\}}{\Gamma; v \vdash \{F\}T\ \ell;\ c : U\{G\}} \quad \text{(Dcl Unknown)}$$

Compared with the standard rule for variable declaration (see (Dcl) on page 36), rule (Dcl Unknown) does not give any information about the variable declared (while in (Dcl), ℓ == $\mathsf{df}(T)$ was added to $(\Gamma, \ell : T; v \vdash \{F\}c : U\{G\})$'s precondition). As a result, the following Hoare triplet (whose command is a single variable declaration) is provable with (Dcl) but not with (Dcl Unknown):

$$\emptyset; v \vdash \{\texttt{true}\}\texttt{Object}\ \ell : \texttt{void}\{\ell == \texttt{null}\}$$

Because we restrict to methods whose preconditions are *receiver splittable*, it is correct to pass unknown parameters to methods.

To model non-deterministic choice, we simply declare boolean variables. Examples include $\mathsf{gen}(r, s \mid s', T)$ where both s and s' can be executed, yet no information is available to decide which branch is to be executed.

In case $\mathsf{gen}(r, s*, T)$, we use a `while` loop to model that s might be executed 0 (if the loop is not entered) or many times (if the loop is entered). Again, we use non-deterministic choice to model that we do not know whether the loop is entered or not.

À la JML, we use `assume` statements to give hints to the verification system when generating the program corresponding to a conditional protocol (case e `?` $s : s$). We cannot use Java's `if` statement directly because expressions occurring in conditional protocols are not valid Java expressions (recall that expressions contain fractional permissions). We use the following Hoare rule to deal with `assume` statements in verification:

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma; v \vdash \{\texttt{true}\}\texttt{assume(}e\texttt{)}\{e\}} \quad \text{(Assume)}$$

Because protocols can model multithreaded programs, $\mathsf{gen}(r, s, T)$ can also generate custom classes extending `Thread` (see cases s `||` s' and `!<`n`>` s and the function $\mathsf{classgen}(C, s, T)$). Generated classes are called from the generated program. In generated classes the receiver is stored in field `rec`. Note that pre and postconditions of the `init` and `run` methods of generated classes should be fulfilled manually.

In the case for heterogeneous parallelism ($\mathsf{gen}(r, s \texttt{ || } s', T)$), two new threads are created and the two protocols are executed in the two threads. Alternatively, we could execute one branch of the protocol in the current thread and simply create one new thread. We choose this solution, however, because it preserves the symmetry of the `||` operator. In the case for homogeneous parallelism ($\mathsf{gen}(r, \texttt{!<}n\texttt{>}\ s, T)$), the generated program creates n new threads, starts them, and then joins them. This is possible, because we restricted the first parameter of `!` to be a known integer. If we used arrays, we could allow integer *variables* to model an unknown level of parallelism.

Finally, we represent class parameters and protocol variables with ghost fields (see $r.w = m(\ell_0, \ldots, \ell_n)$ in case $\mathsf{gen}(r, w \texttt{ = } m, T)$). We need to do so, because conditional protocols can refer to class parameters and protocol variables. We do not formalize ghost fields and use them informally. Formalizing ghost fields would require to extend the syntax of classes, to define an augmented operational semantics (in analogy with the jmlrac compiler [31]). Because class parameters include fractional permissions, which are not a basic Java type, we would in addition need to model fractional permissions with a special class (like JML's model classes).

```
requires true; ensures true;
void checkAdherence(){
 perm p; Collection c;
 i = new Iterator<p,c>;
 Collection c';
 i.init(c');
 boolean b0;
 while(b0){
  i.w = hasNext();
  boolean b1;
  if(b1){
   assume(i.w); Object o = i.next();
   boolean b2;
    if(b2){
     assume(i.p==1);
     boolean b3;
     if(b3){ i.remove(); }
   }
  }
 }
}
```

Figure 7.2: Checking `Iterator`'s Protocol

We believe that, if we had the required extensions, showing the soundness of our technique would be easy by a proof by induction over the structure of protocols. Here, we use the term "soundness" to mean that our technique does not produce false positives. If a generated program cannot be proven correct, then either the verification technique is too weak or the method contracts are incorrect. In both cases, some (client-provided) programs will fail to verify (even if they obey the order on method calls induced by the protocol).

Example: the `Iterator` interface. To check that method contracts of the `Iterator` interface are correct with interface `Iterator`'s protocol (shown in Section 7.3), one has to verify the method shown in Figure 7.2. For convenience, we repeat interface Iterator's contracts in Figure 7.5, where the part of preconditions that correspond to method parameters has been dropped (the initial interface can be found on page 45).

To split preconditions, we use the rules for receiver-splittability defined before. To help the reader understand these rules, we exemplify how we split method `init`'s precondition:

$$\frac{\texttt{mtype}(\texttt{init}, \texttt{Iterator<p, c>}) = \texttt{<>}\ \textit{spec}\ \texttt{void init(Iterator<p, c> this, Collection c)} \quad \texttt{this} \notin \texttt{(c.state<p> * c == iteratee)} \quad \texttt{fv}(\texttt{init}) = \{\texttt{this}\} \subseteq \{\texttt{this}, \texttt{p}, \texttt{iteratee}\}}{\texttt{recs}(\texttt{init * (c.state<p> * c == iteratee)})\ \texttt{in Iterator<p, iteratee>.init}}\ (\text{(Recs Base)})$$

The derivation above means that predicate `init` is the part of method `init`'s precondition relevant to the receiver. The rest (`c.state<p> * c == iteratee`) is discarded because (1) it does not contain `this` and (2) it mentions a method parameter (`c`).

```
interface Iterator<perm p, Collection iteratee>{
   pred ready;                              // prestate for iteration cycle
   pred readyForNext;                       // prestate for next()
   pred readyForRemove<Object element>;     // prestate for remove()
   axiom ready -* iteratee.state<p>;        // stop iterating
   requires init;
   ensures  ready;
   void init(Collection c);
   requires ready;
   ensures  (result -* readyForNext) & (!result -* ready);
   boolean hasNext();
   requires readyForNext;
   ensures  result.state<p> * readyForRemove<result> *
            ((result.state<p> * readyForRemove<result>) -* ready);
   Object next();
   requires readyForRemove<_> * p==1;
   ensures  ready;
   void remove();
}
```

Figure 7.3: Interface `Iterator` with Contracts

Before showing the proof outline for Figure 7.2's program, we show the Hoare rule we use for `while` loops (where F designates the loop invariant):

$$\frac{\Gamma \vdash e, F : \texttt{bool}, \diamond \quad \Gamma; v \vdash \{F \,\&\, e\} c : \texttt{void}\{F\}}{\Gamma; v \vdash \{F\}\texttt{while}\,(e)\{\texttt{inv}\ F;\ c\} : \texttt{void}\{F \,\&\, !e\}} \quad \text{(While)}$$

Figure 7.5 shows the proof outline for Figure 7.2's program (where we use rule (Dcl Unknown) instead of rule (Dcl)). This proof exhibits the relation between interface `Iterator`'s contracts and interface `Iterator`'s protocol. Method `hasNext`'s postcondition uses operator `&` to accommodate the two possible behaviors after `hasNext`: either `hasNext` returned `true` and the left-hand side of `hasNext`'s postcondition applies (point `(1)` of Figure 7.5's proof outline), or `hasNext` returned `false` and the right-hand side of `hasNext`'s postcondition applies (point `(5)`). Similarly, `next`'s postcondition accommodates two cases: either `remove` is called after `next`, in this case the formula containing the wand `-*` is dropped (point `(2)`); or the protocol loops right after `next` (`remove` is not called), in this case modus ponens is used to reestablish the loop's invariant (point `(4)`). Finally, the use of `assume` statements is crucial: assuming that `hasNext` returned `true` is necessary for the proof to go through at point `(1)`, while assuming that the iterator has write access to the collection is necessary for the proof to go through at point `(3)`.

Implementation We implemented the set of rules shown in Figure 7.1 in a tool called *pyrolobus*. Our implementation, examples of protocols, and proofs of the corresponding generated programs are available [62].

```
 { true }
Perm p; Collection c;
 { true }
i=new Iterator<p,c>
 (Because ghost fields v, p, and c have been added to Iterator's definition.)
 { i.init * Perm(i.w,1) * PointsTo(i.p,1,p) * PointsTo(i.c,1,c) }
 (Weakening)
 { i.init * Perm(i.w,1) * PointsTo(i.p,1,p) }
Collection c'; i.init(c');
 { i.ready * Perm(i.w,1) * PointsTo(i.p,1,p) }
boolean b0;
while(b0){
   inv i.ready * Perm(i.w,1) * PointsTo(i.p,1,p);
   { i.ready * Perm(i.w,1) * PointsTo(i.p,1,p) }
 i.w = i.hasNext();
   { PointsTo(i.w,1,w) * PointsTo(i.p,1,p) * ((i.w -* i.readyForNext) & (!i.w -* i.ready)) }
 boolean b1;
 if(b1){
  assume(i.w);
    (Choice on &)              (1)
    { PointsTo(i.w,1,true) * PointsTo(i.p,1,p) * i.readyForNext }
  Object o = i.next();
    { PointsTo(i.w,1,true) * PointsTo(i.p,1,p) * i.readyForRemove<o> *
      o.state<p> * ((o.state<p> * i.readyForRemove<o>) -* i.ready) }
  boolean b2;
  if(b2)){
    assume(i.p==1);
     { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.readyForRemove<o> *
       o.state<1> * ((o.state<1> * i.readyForRemove<o>) -* i.ready) }
    boolean b3;
    if(b3)){
       { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.readyForRemove<o> *
         o.state<1> * ((o.state<1> * i.readyForRemove<o>) -* i.ready) }
       (Weakening)              (2)
       { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.readyForRemove<o> }
     i.remove();               (3)
       { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.ready) }
    } else {
       { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.readyForRemove<o> *
         o.state<1> * ((o.state<1> * i.readyForRemove<o>) -* i.ready) }
       (Modus Ponens)             (4)
       { PointsTo(i.w,1,true) * PointsTo(i.p,1,1) * i.ready }
    }
  } else {
    assume(i.p!=1);
     { PointsTo(i.w,1,true) * PointsTo(i.p,1,p) * i.readyForRemove<o> *
       o.state<p> * ((o.state<p> * i.readyForRemove<o>) -* i.ready) }
     (Modus Ponens)
     { PointsTo(i.w,1,true) * PointsTo(i.p,1,p) * i.ready }
 } else {
  assume(!i.w);
    (Choice on &)              (5)
    { PointsTo(i.w,1,true) * PointsTo(i.p,1,p) * i.ready }
 }
} // end of the loop: in all conditionals, the invariant is reestablished.
 (Weakening)
 { true }
```

Figure 7.4: Proof that interface `Iterator`'s contracts adhere to its protocol

7.6 Related Work and Conclusion

Related Work. Cheon and Perumandla [32] first came up with the idea of specifying protocols of sequential classes with regular expressions. Because we extend their work to deal with protocol variables and parameterized classes, some sequential classes whose protocol cannot be precisely expressed with Cheon and Perumandla's language can now be expressed (such as the `Iterator` example). Jass [8] permits to specify protocols in the style of CSP. An advantage of our approach is that it is more amenable to programmers, because our specification's syntax is easy to grasp.

Both the work of Cheon and Perumandla and Jass support dynamic checking of protocols. We do not provide an implementation for dynamically checking protocols but our goal is different: we use protocols to statically check that method contracts are correct w.r.t. protocols.

Compared to Cheon et al. [32], we do not allow to specify nested call sequences. This forbids to specify that, for example, given two methods m and n, m should call n. We could extend our specifications to allow nested call sequences but it is unclear how to adapt our technique for checking method contracts to such specifications. The reason is that checking method contracts against protocols only makes sense for public protocols (protocols that are used by clients). Nested call sequences, however, are useful for private protocols (protocols that are used by class implementers). Contrary to the work cited, we introduce protocol variables (case w `=` m), we allow to specify optional protocols (case s`?`), to specify conditionals (case e `?` s : s), and to specify parallelism (cases s `||` s, and `!<`n`>` s).

Static checkers such as ESC/Java2 [35] or Boogie [6] permit to statically check user-specified properties of programs, but these tools do not natively support protocols (i.e., protocols can be encoded but cannot be expressed directly).

Conclusion. We provided a concise and intuitive regular expression-like notation to specify protocols of multithreaded Java-like programs that use a variant of generic classes. We presented the semantical foundations of our specification language.

We showed a new technique to show that method contracts are correct w.r.t. a protocol. For this, we generate a program that must be proven correct. If the generated program cannot be proven correct, the method contracts are wrong: some (client-provided) programs will fail to verify, even though they obey the protocol considered. The program generator has been implemented.

Chapter 8

Two Certified Abstractions to Disprove Entailment

In this chapter, we present a new technique to disprove entailment between separation logic formulas. We abstract formulas by the sizes of their possible models and compare these sizes to disprove entailment between formulas. Intuitively, to disprove an entailment $F \vdash G$, it suffices to show that there exists a model (i.e., a heap) of F whose size is smaller than the size of all models of G. We present two different ways to calculate sizes of models, which have different complexity and precision.

Our algorithm is of interest whenever entailment checking is performed, for example in program verifiers. In Chapter 3 to Chapter 5's verification system, entailment checking is necessary to apply the rule (Consequence) (see page 36). In practice, this rule is applied at every step in proof outlines. To apply this rule, program verifiers have to find a formula F' such that (1) $F \vdash F'$ (where F describes the current state) and (2) formula F' matches the next command's precondition. In variants of separation logic containing the magic wand, quantifiers, and permissions (such as Chapter 3's variant of separation logic), finding an appropriate F' is an intricate task. As existing program verifiers [68, 33, 43] do not include the problematic connectors, they do not tackle this issue. In this context, since our algorithm's complexity is low, it can be used to quickly show that a method is not provable, because the maximal size of models of F is smaller than the minimal size of models of the next command's precondition. Importantly, our algorithm works for variants of separation logic that include the problematic connectors.

This chapter is structured as follows: In Section 8.1, we present the variant of separation logic we use. In Section 8.2, we show the two different domains that we use to abstract formulas, i.e., what are sizes. In Section 8.3, we describe the disproving algorithm for intuitionistic separation logic and in Section 8.4, we present the disproving algorithm for classical separation logic. In Section 8.5, we explain why our abstractions are as precise as possible, while being sound. In Section 8.6, we compare the two different abstractions. In Section 8.7, we explain how to extend our algorithm to more expressive fragments of separation logic. In Section 8.8, we discuss the complexity of our algorithms, in Section 8.9, we quickly describe our mechanical proofs, and in Section 8.10, we discuss possible future work. Finally, we discuss related work and conclude in Section 8.11.

Note: The work presented in this chapter is published in the proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO 2009). This work has been done in collaboration with François Bobot (who helped in particular with the Coq proofs) and Alexander J. Summers (who adapted the procedure for the counting model of permissions). This chapter is self contained: notations used in this chapter are not related to notations from other chapters.

8.1 Background

In this chapter, we use a variant of separation logic different from the other chapters of this thesis. On one hand, we do not use Chapter 3's variant of object-oriented separation logic, because object-orientation raises its own problems for the procedure that we describe in this chapter. On the other hand, we use a variant that is more expressive than the variant used in the next chapter. Our choice is motivated as follows: we use the most expressive variant possible, while keeping our procedure simple. That is why object-orientation is not included: it requires a non-trivial extension of the work we present.

We work with permission accounting separation logic based on that of Bornat et al. [21] that generalizes Chapter 3's fractional permissions. We reuse Bornat et al.'s structure of models to abstract over the permission models. A permission model $\mathcal{M}$ consists of:

(a) A commutative semigroup with total binary operator $+_{\mathcal{M}}$.
(b) A minimal permission, m_0, that satisfies $(\forall m)(m_0 +_{\mathcal{M}} m = m)$.
(c) A maximal permission, m_W.
(d) A total order on permissions, $\leq_{\mathcal{M}}$.
(e) A subset of permissions which are defined to be *splittable*.
(f) A pair of functions on permissions, $\mathsf{split1}$ and $\mathsf{split2}$, which define the two parts into which a splittable permission may be split. These functions must satisfy the following property: $(\forall m)(\mathsf{split1}(m) +_{\mathcal{M}} \mathsf{split2}(m) = m)$.

The maximal permission m_W represents full (and exclusive) access to a location, whereas the minimal permission represents no access. Permissions in between these two extremes will allow read access but not write access to a location. Some (in some models, all) permissions are *splittable*: they can be divided into two smaller (in the sense of permission ordering) permissions. In particular, the maximal permission may be split into smaller permissions, none of which allow full access to the location. However, permissions may also be *combined* (by addition): in this way it is possible for a full permission to be regained, if all partial permissions are combined together.

Later, we define heaps that only contain *valid* permissions. Valid permissions are permissions that can be obtained by splitting zero or more times the maximal permission. As a corollary, any valid permission m satisfies the following inequality: $m_0 <_{\mathcal{M}} m \leq_{\mathcal{M}} m_W$. Our model, however, includes invalid permissions, because we need to sum permissions above m_W. In other words, validity of permission makes sense per cell; but as we sum up permissions from all cells, we need invalid permissions. That is why $+_{\mathcal{M}}$ is a total operator. Finally, note that the minimal permission is not valid, because it does not give any right to access the cell considered.

This chapter's results have been established for two concrete permissions models: Boyland's fractional permission model (see Boyland and Bornat et al.'s papers [23, 21] and Chapter 3) and the counting model of Bornat et al. [21].

In the fractional permission model, permissions are rational numbers[1]. The binary operator and the ordering are the usual + and $\leq$ on rationals, the minimal permission is 0 and the maximal permission is 1. All permissions are splittable, and splitting simply divides permissions in half: $\mathsf{split1}(m) = \mathsf{split2}(m) = \frac{m}{2}$.

In the counting model, permissions are originally represented by integers, but we re-encode them in a different manner to allow the use of a lexicographical order. We encode permissions as a pair (n, i), where n is a natural number (in fact, only 0 or 1), and i is an integer. Intuitively, the first of the pair indicates whether this permission can be *split*, to give out further read permissions, while the second indicates a number of read permissions (positive to represent read permissions which have been obtained, and negative to represent those given out). The minimal permission is $(0, 0)$, and the maximal permission is $(1, 0)$. Permissions are added by pairwise addition of their components, and ordered by the lexicographic ordering. A permission is splittable if its first component is 1. The split functions are defined by: $\mathsf{split1}((1, i)) = (1, i - 1)$ and $\mathsf{split2}((1, i)) = (0, 1)$. Contrary to the fractional permission model, the counting model is asymmetric, because the two split functions are asymmetric. Function $\mathsf{split1}$ outputs *source* permissions, that can be further split; while function $\mathsf{split2}$ outputs *unsplittable* permissions. Below, we show how to obtain three read permissions by splitting the maximal permission $(1, 0)$. Among the three read permissions, $(1, -2)$ is a source permission, while the two $(0, 1)$ permissions are unsplittable permissions:

$$(1, 0) \rightsquigarrow (1, -1), (0, 1) \rightsquigarrow (1, -2), (0, 1), (0, 1)$$

We map Bornat's counting permissions onto our pairs model, by the following rules (in which n represents a positive integer):

$$0 \mapsto (1, 0) \qquad n \mapsto (1, -n) \qquad -n \mapsto (0, n)$$

We study both intuitionistic and classical separation logic [67]. Both variants contains pure (heap independent) and spatial (heap dependent) formulas:

n	$\in$	$\mathbb{N}$		
a, v	$\in$	$n \mid n + n \mid n - n \mid \ldots$	addresses and values	
m			abstract permissions	
Π	::=	$a = a \mid a \neq a \mid \texttt{true} \mid \ldots$	pure formulas	
Σ^I	::=	$a \overset{m}{\hookrightarrow} v$	intuitionistic spatial formulas	(m must be valid)
Σ^C	::=	$\texttt{emp} \mid a \overset{m}{\mapsto} v$	classical spatial formulas	(m must be valid)

In intuitionistic separation logic, the atomic spatial formula is the *harpoon* $a \overset{m}{\hookrightarrow} v$ while in classical separation logic, the atomic spatial formula is the *points-to* predicate $a \overset{m}{\mapsto} v$. The harpoon $a \overset{m}{\hookrightarrow} v$'s is very similar to Chapter 3's `PointsTo` predicate. Firstly, $a \overset{m}{\hookrightarrow} v$ asserts that the heap contains a cell at address a with content v. Secondly, $a \overset{m}{\hookrightarrow} v$ asserts permission m to the cell at address a. If $m = m_W$, $a \overset{m}{\hookrightarrow} v$ asserts write and read

[1] In Chapter 3, the semantics of fractional permissions was included in the set of rational numbers in $(0, 1]$ (see the definition of $[\![.]\!]$ for specification values on page 29). This is inadequate for this chapter, because we sum fractional permissions and end up with fractions whose semantics is above 1. Chapter 3's permissions are now called *valid* permissions.

authorization to the cell at address a, otherwise it asserts *readonly* authorization. The points-to predicate $a \overset{m}{\mapsto} v$ has the same meaning as the harpoon but it enforces that the heap contains *only* the cell at address a. In addition, classical separation logic includes the predicate `emp` that denotes an empty heap. Intuitionistic (respectively classical) separation logic is obtained by taking Σ to be Σ^I (respectively Σ^C) below:

$$A, B \quad ::= \quad \Pi \mid \Sigma \mid A \star A \mid A \wedge A \mid A \vee A \quad \text{formulas}$$

In $A \star B$, $\star$ is the *separating conjunction* (like Chapter 3's $*$). $A \star B$ represents a heap consisting of two *separate* subheaps A and B. $A \wedge B$ is the usual logical conjunction, which represents two different views of a heap (like Chapter 3's `&`). We do not include the standard implication $\Rightarrow$ as this keeps the approach simple by avoiding bunched contexts in the proof system [85]. The fact that most separation-logic-based program verifiers do not include a standard implication (see the implementations of program verifiers based on separation logic [15, 68, 43] and Chapter 2 to Chapter 5's verification system for Java-like programs) supports our choice. In addition, we use a language without variables, but we describe how variables impact our algorithm in Section 8.7.3.

Because our algorithms are defined in Coq [36], we represent heaps as lists (which are well supported by Coq's standard library). A list entry is a triplet of an address, a valid permission, and a value:

$$\begin{array}{rcll} c & ::= & (a, m, v) & \text{heap cells} \\ h & ::= & [] \mid c :: h & \text{heaps} \end{array}$$

We define a projection operator to extract a value from a cell: $\mathsf{val}(a, m, v) = v$ and we write $h[a]$ to denote h's set of heap cells whose address are a:

$$\begin{array}{rcl} [][a] & \triangleq & \emptyset \\ ((a, m, v) :: h)[a'] & \triangleq & \begin{cases} \{(a, m, v)\} \cup h[a] & \text{if } a = a' \\ h[a] & \text{otherwise} \end{cases} \end{array}$$

We write $h(a)$ to denote the sum of permissions to a occurring in h:

$$\begin{array}{rcl} [](a) & \triangleq & m_0 \\ ((a, m, v) :: h)(a') & \triangleq & \begin{cases} m +_{\mathcal{M}} h(a) & \text{if } a = a' \\ h(a) & \text{otherwise} \end{cases} \end{array}$$

Conjunction of two heaps is simply list concatenation (written $h@h'$) and compatibility of heaps is standard[2]. As usual, two heaps are compatible if the sum of permissions to each cell is valid and if the two heaps coincide on values:

$$h \# h' \triangleq (\forall a) \left(\begin{array}{l} h(a) +_{\mathcal{M}} h'(a) \leq m_W \text{ and} \\ (\forall c \in h[a], \forall c' \in h'[a])(\mathsf{val}(c) = \mathsf{val}(c')) \end{array} \right)$$

[2] In Chapter 3, conjunction of heaps was written $h{*}h'$ (see page 26) while compatibility of heaps was written – as in this chapter – $h\#h'$ (see page 26).

The semantics of formulas is standard. In particular, the semantics of pure formulas is left unaxiomatized:

$$
\begin{array}{lll}
h \models \Pi & \text{iff} & \mathsf{oracle}(\Pi) \\
h \models \mathtt{emp} & \text{iff} & h = [] \\
h \models a \overset{m}{\hookrightarrow} v & \text{iff} & (\exists h')(h = (a, m, v) @ h') \\
h \models a \overset{m}{\mapsto} v & \text{iff} & h = (a, m, v) \\
h \models A \star B & \text{iff} & (\exists h_1, h_2)(h_1 \# h_2, h = h_1 @ h_2, h_1 \models A, \text{ and } h_2 \models B) \\
h \models A \wedge B & \text{iff} & h \models A \text{ and } h \models B \\
h \models A \vee B & \text{iff} & h \models A \text{ or } h \models B
\end{array}
$$

We write $A \vdash B$ to denote that A entails B. Figure 8.1 shows $\vdash$'s definition, i.e., the (standard) proof system. This proof system is very close to the proof system we defined for Chapter 3's verification system (see page 32). The only non-standard rules are (Splitting) and (Merging) that lift permission splitting to formulas[3]:

$$\frac{}{A \vdash A}\ (\text{Id}) \qquad \frac{A \vdash B}{A, C \vdash B}\ (\text{Weak}) \qquad \frac{\mathsf{oracle}(\Pi)}{\mathtt{true} \vdash \Pi}\ (\text{Oracle})$$

$$\frac{A_1 \vdash B_1 \qquad A_2 \vdash B_2}{A_1, A_2 \vdash B_1 \star B_2}\ (\star \text{ Intro}) \qquad \frac{A \vdash B_1 \star B_2 \qquad C, B_1, B_2 \vdash D}{A, C \vdash D}\ (\star \text{ Elim})$$

$$\frac{A \vdash B_1 \qquad A \vdash B_2}{A \vdash B_1 \wedge B_2}\ (\wedge \text{ Intro}) \qquad \frac{A \vdash B_1 \wedge B_2}{A \vdash B_1}\ (\wedge \text{ Elim 1}) \qquad \frac{A \vdash B_1 \wedge B_2}{A \vdash B_2}\ (\wedge \text{ Elim 2})$$

$$\frac{A \vdash B_1}{A \vdash B_1 \vee B_2}\ (\vee \text{ Intro 1}) \qquad \frac{A \vdash B_2}{A \vdash B_1 \vee B_2}\ (\vee \text{ Intro 2})$$

$$\frac{\vee \text{ does not occur in } A}{A \vdash \mathsf{split1}(A) \star \mathsf{split2}(A)}\ (\text{Splitting}) \qquad \frac{\vee \text{ does not occur in } A}{\mathsf{split1}(A) \star \mathsf{split2}(A) \vdash A}\ (\text{Merging})$$

Figure 8.1: Proof System

In the proof rules (Splitting) and (Merging), we use split1 and split2 (defined below) to lift permission splitting to formula splitting. The conditions on the proof rules for splitting and merging ensure that no disjunctions occur within the formulas concerned. This is because splitting and merging is unsound for disjunctions (and formulas with the magic wand). As we only split formulas without these connectives, we need not define those cases for the definition of splitting formulas:

[3] This means these rules could be used to prove the axiom for groups that we introduce in `group`'s desugaring (see page 53). We have shown in our previous work [53] how to do that for Chapter 3's variant of separation logic. What we do now is similar but simpler, because we do not have abstract predicates.

$$
\begin{array}{lcl}
\mathsf{split1}(\Pi) & \triangleq & \Pi \\
\mathsf{split1}(\mathtt{emp}) & \triangleq & \mathtt{emp} \\
\mathsf{split1}(a \overset{m}{\hookrightarrow} v) & \triangleq & a \overset{\mathsf{split1}(m)}{\hookrightarrow} v \\
\mathsf{split1}(a \overset{m}{\mapsto} v) & \triangleq & a \overset{\mathsf{split1}(m)}{\mapsto} v \\
\mathsf{split1}(A \star B) & \triangleq & \mathsf{split1}(A) \star \mathsf{split1}(B) \\
\mathsf{split1}(A \wedge B) & \triangleq & \mathsf{split1}(A) \wedge \mathsf{split1}(B) \\
\\
\mathsf{split2}(\Pi) & \triangleq & \Pi \\
\mathsf{split2}(\mathtt{emp}) & \triangleq & \mathtt{emp} \\
\mathsf{split2}(a \overset{m}{\hookrightarrow} v) & \triangleq & a \overset{\mathsf{split2}(m)}{\hookrightarrow} v \\
\mathsf{split2}(a \overset{m}{\mapsto} v) & \triangleq & a \overset{\mathsf{split2}(m)}{\mapsto} v \\
\mathsf{split2}(A \star B) & \triangleq & \mathsf{split2}(A) \star \mathsf{split2}(B) \\
\mathsf{split2}(A \wedge B) & \triangleq & \mathsf{split2}(A) \wedge \mathsf{split2}(B)
\end{array}
$$

The proof system is sound w.r.t. to the semantics above:

Theorem 7 (Soundness of the proof system)**.** *If $A \vdash B$, then for all h, $h \models A$ implies $h \models B$.*

Because we use it later, we spell out Theorem 7's contraposition:

Theorem 8 (Contraposition of the proof system's soundness)**.** *If there exists h such that $h \models A$ and $h \not\models B$, then $A \not\vdash B$.*

8.2 Domain of Abstraction: Sizes

We abstract models of separation logic formulas (i.e., heaps) by their size. We use two abstractions of different precisions: the first abstraction is a *whole heap* abstraction, while the second abstraction is a *per cell* abstraction. The two different abstractions have a different trade-off between complexity and precision. The whole heap abstraction is imprecise but has a low complexity both in time (see Section 8.8) and space. The per cell abstraction is precise but has a higher complexity both in time (see Section 8.8) and space.

For the whole heap abstraction, the size of a heap is simply the sum of all permissions occurring in this heap:

$$\mathsf{size}_{\mathcal{M}}([]) \triangleq m_0 \qquad \mathsf{size}_{\mathcal{M}}((a, m, v) :: h) \triangleq m +_{\mathcal{M}} \mathsf{size}_{\mathcal{M}}(h)$$

For the per cell abstraction, the size of a heap is a *permission table* (like Chapter 3's permission tables on page 26) i.e., the corresponding heap without values:

$$
\begin{array}{rcll}
\mathcal{P}_c & ::= & (a, m) & \text{permission table cells} \\
\mathcal{P} & ::= & [] \mid \mathcal{P}_c :: \mathcal{P} & \text{permission tables}
\end{array}
$$

$$\mathsf{size}_{\mathcal{P}}([]) \triangleq [] \qquad \mathsf{size}_{\mathcal{P}}((a, m, v) :: h) \triangleq (a, m) :: \mathsf{size}_{\mathcal{P}}(h)$$

As for heaps, we write $\mathcal{P}(a)$ to denote the sum of permissions to a occurring in $\mathcal{P}$. Sizes are equipped with an order. For the whole heap abstraction, we use the order on

permissions $\leq_{\mathcal{M}}$, while for the per cell abstraction, we use an order on permission tables $\leq_{\mathcal{P}}$:

$$\mathcal{P} \leq_{\mathcal{P}} \mathcal{P}' \triangleq (\forall a)(\mathcal{P}(a) \leq \mathcal{P}'(a))$$

Sizes are equipped with minimum and maximum functions. For the whole heap abstraction, we use standard definitions according to the corresponding order on permissions $\leq_{\mathcal{M}}$:

$$\mathsf{smin}_{\mathcal{M}}(m, m') \triangleq \begin{cases} m & \text{iff} \quad m \leq_{\mathcal{M}} m' \\ m' & \text{otherwise} \end{cases} \qquad \mathsf{smax}_{\mathcal{M}}(m, m') \triangleq \begin{cases} m' & \text{iff} \quad m \leq_{\mathcal{M}} m' \\ m & \text{otherwise} \end{cases}$$

For the per cell abstraction, we use definitions that merge permission tables point-wise:

$$\mathsf{smin}_{\mathcal{P}}(\mathcal{P}, \mathcal{P}') \triangleq \{\mathcal{P}'' \mid (\forall a)(\mathcal{P}''(a) = \mathsf{smin}_{\mathcal{M}}(\mathcal{P}(a), \mathcal{P}'(a)))\}$$
$$\mathsf{smax}_{\mathcal{P}}(\mathcal{P}, \mathcal{P}') \triangleq \{\mathcal{P}'' \mid (\forall a)(\mathcal{P}''(a) = \mathsf{smax}_{\mathcal{M}}(\mathcal{P}(a), \mathcal{P}'(a)))\}$$

From now on, we subscript functions when we speak about a concrete abstraction while we use functions without subscripts when we describe properties of both abstractions.

8.3 The Disproving Algorithm for Intuitionistic Separation Logic

To disprove entailment between formulas in intuitionistic separation logic, we search for models with bounded sizes. Formally, we later define functions min and max that must satisfy the following property:

Theorem 9 (Property of min and max)**.** *If $h \models A$, then $\mathsf{min}(A) \leq \mathsf{size}(h)$ and there exist h_s such that, $h_s \models A$ and $\mathsf{size}(h_s) \leq \mathsf{max}(A)$.*

Intuitively, Theorem 9 states that, given a satisfiable formula A (i.e., there exists h such that $h \models A$), there exists a model h_s of A whose size is in between $\mathsf{min}(A)$ and $\mathsf{max}(A)$. Then, we use min and max to disprove entailment as follows:

Theorem 10 (min and max gives a disproving algorithm)**.** *If A is satisfiable and $\mathsf{min}(B) \leq \mathsf{max}(A)$ does* not *hold, then $A \not\vdash B$.*

Proof. Because A is satisfiable, there exists h such that $h \models A$. Then, by Theorem 9, it follows that there exists h_s such that $h_s \models A$ and $\mathsf{size}(h_s) \leq \mathsf{max}(A)$.

Now, suppose $h_s \models B$. By Theorem 9, it follows that $\mathsf{min}(B) \leq \mathsf{size}(h_s)$. From $\mathsf{size}(h_s) \leq \mathsf{max}(A)$, by transitivity, it follows that $\mathsf{min}(B) \leq \mathsf{max}(A)$. This contradicts, however, the hypothesis that $\mathsf{min}(B) \leq \mathsf{max}(A)$ does not hold. Hence, $h_s \not\models B$.

From $h_s \models A$ and $h_s \not\models B$, by Theorem 8, it follows that $A \not\vdash B$. □

The reader might wonder why we formulate Theorem 10 with a negation on the $\leq$ operator instead of using a $<$ operator. The reason is that, for permission tables, $\neg(\mathcal{P} \leq_{\mathcal{P}} \mathcal{P}')$ is *not* equivalent to $\mathcal{P}' <_{\mathcal{P}} \mathcal{P}$ (where $<_{\mathcal{P}}$ would be defined in a way similar to $\leq_{\mathcal{P}}$).

In the remainder of this section, we show definitions of min and max for the whole heap abstraction and the per cell abstraction. As Section 8.9 discusses, these definitions satisfy Theorem 9.

8.3.1 Whole Heap Abstraction for Intuitionistic Separation Logic

min and max are defined as follows:

$$\begin{array}{rcl} \mathsf{min}(\Pi) & \triangleq & m_0 \\ \mathsf{min}(a \overset{m}{\hookrightarrow} v) & \triangleq & m \\ \mathsf{min}(A \star B) & \triangleq & \mathsf{min}(A) +_{\mathcal{M}} \mathsf{min}(B) \\ \mathsf{min}(A \wedge B) & \triangleq & \mathsf{smax}_{\mathcal{M}}(\mathsf{min}(A), \mathsf{min}(B)) \\ \mathsf{min}(A \vee B) & \triangleq & \mathsf{smin}_{\mathcal{M}}(\mathsf{min}(A), \mathsf{min}(B)) \end{array} \qquad \begin{array}{rcl} \mathsf{max}(\Pi) & \triangleq & m_0 \\ \mathsf{max}(a \overset{m}{\hookrightarrow} v) & \triangleq & m \\ \mathsf{max}(A \star B) & \triangleq & \mathsf{max}(A) +_{\mathcal{M}} \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) & \triangleq & \mathsf{max}(A) +_{\mathcal{M}} \mathsf{max}(B) \\ \mathsf{max}(A \vee B) & \triangleq & \mathsf{smax}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B)) \end{array}$$

8.3.2 Per Cell Abstraction for Intuitionistic Separation Logic

min and max are defined as follows:

$$\begin{array}{rcl} \mathsf{min}(\Pi) & \triangleq & [] \\ \mathsf{min}(a \overset{m}{\hookrightarrow} v) & \triangleq & (a, m) :: [] \\ \mathsf{min}(A \star B) & \triangleq & \mathsf{min}(A) @ \mathsf{min}(B) \\ \mathsf{min}(A \wedge B) & \triangleq & \mathsf{smax}_{\mathcal{P}}(\mathsf{min}(A), \mathsf{min}(B)) \\ \mathsf{min}(A \vee B) & \triangleq & \mathsf{smin}_{\mathcal{P}}(\mathsf{min}(A), \mathsf{min}(B)) \end{array} \qquad \begin{array}{rcl} \mathsf{max}(\Pi) & \triangleq & [] \\ \mathsf{max}(a \overset{m}{\hookrightarrow} v) & \triangleq & (a, m) :: [] \\ \mathsf{max}(A \star B) & \triangleq & \mathsf{max}(A) @ \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) & \triangleq & \mathsf{smax}_{\mathcal{P}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) & \triangleq & \mathsf{smax}_{\mathcal{P}}(\mathsf{max}(A), \mathsf{max}(B)) \end{array}$$

Although max's definitions are very similar for the two abstractions, there is one case where they differ: in case $\wedge$ of max. In analogy with the per cell abstraction, one could expect $\mathsf{max}(A \wedge B)$ to be $\mathsf{smax}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B))$ in the whole heap abstraction. To see why this is unsound, however, one can consider a formula where the right and left hand sides of a $\wedge$ represent separate parts of the heap such as $42 \overset{1}{\hookrightarrow} \_ \wedge 47 \overset{1}{\hookrightarrow} \_$ (where $\_$ denotes irrelevant values).

8.4 The Disproving Algorithm for Classical Separation Logic

To disprove entailment between formulas in classical separation logic, we pursue a slightly different goal than for intuitionistic separation logic: instead of exhibiting *one* model whose size is bounded, we search for bounds on the size of *all* models.

Because we allow pure formulas, which do not constrain the size of heaps modelling them, we cannot always find an upper-bound on the size of heaps. Hence we add a distinguished "maximal" size that we write ∞ (both for the whole heap and for the per cell abstraction). The meaning of ∞ is axiomatized as follows:

$$(\forall m)(m \leq_{\mathcal{M}} \infty) \qquad (\forall \mathcal{P})(\mathcal{P} \leq_{\mathcal{P}} \infty)$$

Theorem 11 (Property of min and max)**.** *If $h \models A$, then $\mathsf{min}(A) \leq \mathsf{size}(h) \leq \mathsf{max}(A)$.*

Then, we use min and max to disprove entailment as follows:

Theorem 12 (min and max gives a disproving algorithm)**.** *If A is satisfiable and $\mathsf{min}(B) \leq \mathsf{max}(A)$ does not hold, then $A \not\vdash B$.*

Proof. Similar to the proof of Theorem 10. □

In the remainder of this section, we give definitions of min and max for the whole heap abstraction and the per cell abstraction. As Section 8.9 discusses, these definitions satisfy Theorem 11. For min, we only show its definition in case of a formula `emp`; for other cases min's definition is similar to the intuitionistic case (see Sections 8.3.1 and 8.3.2).

8.4.1 Whole Heap Abstraction for Classical Separation Logic

min and max are defined as follows:

$$\mathsf{min}(\mathtt{emp}) \triangleq m_0 \qquad \begin{array}{rcl} \mathsf{max}(\Pi) & \triangleq & \infty \\ \mathsf{max}(\mathtt{emp}) & \triangleq & m_0 \\ \mathsf{max}(a \overset{m}{\mapsto} v) & \triangleq & m \\ \mathsf{max}(A \star B) & \triangleq & \mathsf{max}(A) +_{\mathcal{M}} \mathsf{max}(B) \\ \mathsf{max}(A \wedge B) & \triangleq & \mathsf{smin}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) & \triangleq & \mathsf{smax}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B)) \end{array}$$

As as example, consider the following computations of the maximum and minimum of two formulas (for the fractional permissions model):

$$\begin{array}{rl} & \mathsf{max}((42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) \vee ((42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_) \wedge (47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_))) \\ = & \mathsf{smax}_{\mathcal{M}}(1 +_{\mathcal{M}} 1/2, \mathsf{smin}_{\mathcal{M}}(\mathsf{max}(42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_), \mathsf{max}(47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_))) \\ = & \mathsf{smax}_{\mathcal{M}}(\frac{3}{2}, \mathsf{smin}_{\mathcal{M}}(\frac{7}{4}, \frac{7}{4})) = \frac{7}{4} \end{array}$$

$$\mathsf{min}(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) = 1 +_{\mathcal{M}} \tfrac{1}{2} +_{\mathcal{M}} \tfrac{1}{2} = 2$$

Now, because $2 \leq_{\mathcal{M}} 7/4$ does not hold (note that, as Theorem 11 requires, the formula on the left-hand side of $\vdash$ is satisfiable), our algorithm deduces:

$$(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) \vee ((42 \overset{1}{\mapsto} \_ \star 47 \overset{3/4}{\mapsto} \_) \wedge (47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/4}{\mapsto} \_ \star 42 \overset{1}{\mapsto} \_)) \not\vdash 42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_$$

We now compare max's definition for $\star$ and $\wedge$ in both semantics (similar remarks apply for the per cell abstraction).

- In the classical semantics, we have $\mathsf{max}(A \star B) = \mathsf{max}(A) +_{\mathcal{M}} \mathsf{max}(B)$, while we have $\mathsf{max}(A \wedge B) = \mathsf{smin}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This reflects the intuition that, in $A \star B$, A and B represents *separate* heaps, while in $A \wedge B$, A and B represent different *views* of the *same* heap.
- In the intuitionistic semantics, however, we have $\mathsf{max}(A \star B) = \mathsf{max}(A \wedge B) = \mathsf{max}(A) +_{\mathcal{M}} \mathsf{max}(B)$. This reflects that, both in $A \star B$ *and* in $A \wedge B$, A and B must (for $\star$) or may (for $\wedge$) represent separate parts of the heap.

8.4.2 Per Cell Abstraction for Classical Separation Logic

min and max are defined as follows:

$$\mathsf{min}(\mathtt{emp}) \triangleq [] \qquad \begin{array}{rcl} \mathsf{max}(\Pi) & \triangleq & \infty \\ \mathsf{max}(\mathtt{emp}) & \triangleq & [] \\ \mathsf{max}(a \overset{m}{\mapsto} v) & \triangleq & (a, m) :: [] \\ \mathsf{max}(A \star B) & \triangleq & \begin{cases} \mathsf{max}(A) @ \mathsf{max}(B) & \text{if } \mathsf{max}(A) \neq \infty \text{ and} \\ & \quad \mathsf{max}(B) \neq \infty \\ \infty & \text{otherwise} \end{cases} \\ \mathsf{max}(A \wedge B) & \triangleq & \mathsf{smin}_{\mathcal{P}}(\mathsf{max}(A), \mathsf{max}(B)) \\ \mathsf{max}(A \vee B) & \triangleq & \mathsf{smax}_{\mathcal{P}}(\mathsf{max}(A), \mathsf{max}(B)) \end{array}$$

In this case, the definitions of max are similar in both abstractions.

8.4.3 On ∞

We elaborate here on the consequence of our use of ∞ above for the whole heap abstraction (similar comments apply to the per cell abstraction).

The $\star$ and $\vee$ connectives have a different behavior w.r.t. ∞ than $\wedge$. To see why, consider the following properties of $+_{\mathcal{M}}, \mathsf{smin}_{\mathcal{M}}$, and $\mathsf{smax}_{\mathcal{M}}$ w.r.t. ∞:

$$(\forall m)(m +_{\mathcal{M}} \infty = \infty +_{\mathcal{M}} m = \infty)$$
$$(\forall m)(\mathsf{smin}_{\mathcal{M}}(m, \infty) = \mathsf{smin}_{\mathcal{M}}(\infty, m) = m)$$
$$(\forall m)(\mathsf{smax}_{\mathcal{M}}(m, \infty) = \mathsf{smax}_{\mathcal{M}}(\infty, m) = \infty)$$

Because of the properties above, both the $\star$ and $\vee$ operators "spread" ∞: for $op \in \{\star, \vee\}$, $\mathsf{max}(A \ op \ B) = \infty$ if $\mathsf{max}(A) = \infty$ or $\mathsf{max}(B) = \infty$. The $\wedge$ operator, however, stops ∞ from spreading: $\mathsf{max}(A \wedge B) \neq \infty$ if $\mathsf{max}(A) \neq \infty$ or $\mathsf{max}(B) \neq \infty$.

Because the variants of separation logic used in program verifiers [15, 43] impose pure formulas and spatial formulas to be $\wedge$-conjoined at the top level, max would never return ∞ in these variants.

8.5 Precision

Ideally, min and max's definitions need to be as tight as possible, while retaining soundness (Theorems 9 and 11). We exemplify this for both variants of the logic. Our explanations focus solely on the whole heap abstraction but our remarks apply to both abstractions.

In the intuitionistic semantics, we have $\mathsf{max}(A \wedge B) = \mathsf{max}(A) + \mathsf{max}(B)$. At first sight, this might appear too loose, but the semantics of the harpoon requires this definition for soundness. To see why, one can consider a formula in which the right and left hand sides of a $\wedge$ represent separate parts of the heap, such as $42 \overset{1}{\hookrightarrow} \_ \wedge 47 \overset{1}{\hookrightarrow} \_$. Models of this formula must have a size greater or equal to 2, because $\wedge$'s operands represent separate parts of the heap.

In the classical semantics, we have $\mathsf{max}(A \wedge B) = \mathsf{smin}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This definition is tight because, if $\mathsf{max}(A) < \mathsf{max}(B)$, by $\wedge$'s semantics, it follows that any model h of B whose size is greater than $\mathsf{max}(A)$ cannot be a model of $A \wedge B$, because h cannot satisfy A (and conversely). In the classical semantics, we have $\mathsf{max}(A \vee B) = \mathsf{smax}_{\mathcal{M}}(\mathsf{max}(A), \mathsf{max}(B))$. This definition is also tight because, if $\mathsf{max}(A) < \mathsf{max}(B)$, by $\vee$'s semantics, it follows that a model of B can be a model of $A \vee B$ (and conversely).

8.6 Comparison Between the Two Abstractions

The per cell abstraction is more powerful (or discriminative) than the whole heap abstraction. In another words, whenever the whole heap abstraction concludes that a formula does not entail another formula, the per cell abstraction concludes the same (but not the other way round). Formally, this statement is expressed by the following theorem:

Theorem 13 (The Per Cell Abstraction is More Powerful)**.** *If the whole heap abstraction concludes $A \not\vdash B$, then the per cell abstraction concludes $A \not\vdash B$.*

Example. Here, we show an example (in the classical semantics) where the powerfulness of the per cell abstraction is necessary to obtain the desired result. Concretely, we show A and B such that the whole heap abstraction cannot conclude $A \vdash B$, while the per cell abstraction does conclude $A \nvdash B$.

We define:

$$A \triangleq 42 \overset{1/2}{\mapsto} \_ \star 47 \overset{1}{\mapsto} \_ \qquad B \triangleq 42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_$$

In the whole heap abstraction, we have:

$$\mathsf{max}(42 \overset{1/2}{\mapsto} \_ \star 47 \overset{1}{\mapsto} \_) = 1/2 +_{\mathcal{M}} 1 = 3/2 \qquad \mathsf{min}(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) = 1/2 +_{\mathcal{M}} 1 = 3/2$$

Because $3/2 \leq 3/2$ holds, we cannot conclude that $42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_ \nvdash 42 \overset{1/2}{\mapsto} \_ \star 47 \overset{1}{\mapsto} \_$. Now, consider the computations of min and max in the per cell abstraction:

$$\mathsf{max}(42 \overset{1/2}{\mapsto} \_ \star 47 \overset{1}{\mapsto} \_) = (42, 1/2) :: (47, 1) \qquad \mathsf{min}(42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_) = (42, 1) :: (47, 1/2)$$

The following holds:

$$\big((42, 1) :: (47, 1/2)\big)(42) = 1 \not\leq 1/2 = \big((42, 1/2) :: (47, 1)\big)(42)$$

As a consequence, by Theorem 11, we can conclude:

$$42 \overset{1/2}{\mapsto} \_ \star 47 \overset{1}{\mapsto} \_ \nvdash 42 \overset{1}{\mapsto} \_ \star 47 \overset{1/2}{\mapsto} \_$$

8.7 Extension to Other Operators of Separation Logic

In this section, we review extensions of our algorithm for which our results are partial. Our algorithm has been extended to deal with the magic wand but our extension might be loose (w.r.t. precision) in the intuitionistic case (Section 8.7.1), soundness of min and max for quantifiers has been checked only on pen and paper (Section 8.7.2), and variables have been integrated only with the whole heap abstraction (Section 8.7.3).

8.7.1 The magic wand $-\!\!\star$

The magic wand $-\!\!\star$ matches the resource conjunction $\star$, in the sense that the modus ponens law is satisfied: $A \star (A -\!\!\star B)$ implies B. The magic wand's semantics quantifies over models of the wand's left-hand side:

$$h \models A -\!\!\star B \quad \text{iff} \quad (\forall h')((h' \models A \text{ and } h' \# h) \text{ implies } h' @ h \models B)$$

The following rules are added to the proof system to handle the wand:

$$\frac{A, B_1 \vdash B_2}{A \vdash B_1 -\!\!\star B_2}\ (-\!\!\star \text{ Intro}) \qquad \frac{A \vdash C_1 -\!\!\star C_2 \qquad B \vdash C_1}{A, B \vdash C_2}\ (-\!\!\star \text{ Elim})$$

In both semantics, we have mechanically verified that our algorithm can be extended to deal with the magic wand. Because min and max are now mutually recursive, min, like max, can return ∞.

Intuitionistic Semantics. In this case, the definitions of min and max from Section 8.3.1 for the whole heap abstraction are completed as follows:

$$\mathsf{min}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{min}(B) & \text{if } \mathsf{min}(B) \neq \infty \\ m_0 & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{max}(B) & \text{if } \mathsf{max}(B) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

For the per cell abstraction, definitions of min and max from Section 8.3.2 are completed as follows:

$$\mathsf{min}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{min}(B) & \text{if } \mathsf{min}(B) \neq \infty \\ [] & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{max}(B) & \text{if } \mathsf{max}(B) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

Classical Semantics. In this case, the definitions of min and max from Section 8.4.1 for the whole heap abstraction are completed as follows:

$$\mathsf{min}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{min}(B) -_{\mathcal{M}} \mathsf{max}(A) & \text{if } \mathsf{min}(B) \neq \infty \text{ and } \mathsf{max}(A) \neq \infty \\ m_0 & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{max}(B) -_{\mathcal{M}} \mathsf{min}(A) & \text{if } \mathsf{max}(B) \neq \infty \text{ and } \mathsf{min}(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

For the per cell abstraction, definitions of min and max from Section 8.4.2 are completed as follows:

$$\mathsf{min}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{min}(B)@ -_{\mathcal{P}} (\mathsf{max}(A)) & \text{if } \mathsf{min}(B) \neq \infty \text{ and } \mathsf{max}(A) \neq \infty \\ [] & \text{otherwise} \end{cases}$$

$$\mathsf{max}(A \mathrel{-\!\!\star} B) \quad = \quad \begin{cases} \mathsf{max}(B)@ -_{\mathcal{P}} (\mathsf{min}(A)) & \text{if } \mathsf{max}(B) \neq \infty \text{ and } \mathsf{min}(A) \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

where $-_{\mathcal{P}}$ pointwisely applies the unary $-_{\mathcal{M}}$ operator to permission tables:

$$-_{\mathcal{P}}([]) = []$$
$$-_{\mathcal{P}}((a, m) :: \mathcal{P}) = (a, -_{\mathcal{M}}(m)) :: -_{\mathcal{P}}(\mathcal{P})$$

In both semantics, for the definitions above to be sound, we need to assume that formulas appearing on the left-hand side of the magic wand are satisfiable. While this is harmful from a theoretical point of view, examples where the wand is used (see the `Iterator` interface in Section 3.6.2 and the class `DatabaseUpdater` in Section 4.6.2) suggest this is a plausible assumption for program verifiers (we do not include examples from the literature because the wand is scarcely used).

We did not include the magic wand in Sections 8.3 and 8.4, because we believe the precision of the definitions of min and max in the intuitionistic semantics can be enhanced (i.e., by being similar to the classical semantics's definitions). This is, however, a non-trivial task. In a nutshell, the universal quantification in the magic wand's semantics and the way Theorem 9 is formulated (i.e., with an existential quantifier) prevent a simple proof by induction.

8.7.2 Quantification

We distinguish between quantification on values and permissions. We add the following formulas to Section 8.1's language:

$$A, B \quad ::= \quad \ldots \mid \exists v.\, A \mid \exists m.\, A \mid \forall v.\, A \mid \forall m.\, A$$

The semantics is standard and we omit the corresponding proof rules which are also standard. To define min and max, we introduce a "very small permission" ϵ. The meaning of ϵ is axiomatized as follows:

$$(\forall m)(m <_{\mathcal{M}} m +_{\mathcal{M}} \epsilon)$$

For both abstractions min and max are defined as follows:

$$\begin{array}{ll} \mathsf{min}(\exists v.\, A) = \mathsf{min}(A) & \mathsf{min}(\exists m.\, A) = \mathsf{min}(A[\epsilon/m]) \\ \mathsf{max}(\exists v.\, A) = \mathsf{max}(A) & \mathsf{max}(\exists m.\, A) = \mathsf{max}(A[m_W/m]) \\ \mathsf{min}(\forall v.\, A) = \mathsf{min}(A) & \mathsf{min}(\forall m.\, A) = \mathsf{min}(A[m_W/m]) \\ \mathsf{max}(\forall v.\, A) = \mathsf{max}(A) & \mathsf{max}(\forall m.\, A) = \mathsf{max}(A[m_W/m]) \end{array}$$

8.7.3 Variables

In real-world variants of separation logic, variables are used. For the whole heap abstraction, variables (including quantification over variables) do not create any extra difficulties for our work. For the per cell abstraction, it is unclear at this stage how variables can be handled in an optimal way.

8.8 Complexity

The complexity of our techniques is linear for the whole heap abstraction, and $O(n(\log n))$ for the per cell abstraction (where n is the input formula's size, defined in a standard way). To our knowledge, there is no complexity result for entailment checking in separation logic. We believe the complexity of our disproving algorithm is low enough to be useful when entailment must be checked quickly. This is supported by the fact that integrating both the magic wand and quantifiers do not increase our algorithm's complexity, while they typically increase by orders of magnitude the complexity of model-checking or validity-checking [30, 27].

8.9 Soundness

All theorems from Section 8.1, 8.3, and 8.4 have been mechanically checked with Coq [36]. Addition of the magic wand (Section 8.7.1) has also been mechanically checked. Proofs about the proof system (Theorems 7 and 8); properties of permissions, heaps, permissions tables etc. are 3090 lines long. The proofs of Theorems 9, 10, 11, and 12 are 620 lines long. The proofs have been engineered so that certified implementations of the two abstractions could be extracted. Proof scripts are available online [62].

Proofs for quantifiers (Section 8.7.2) have been checked only on pen and paper.

8.10 Future Work

This chapter's procedure can be extended in two ways. First, it can be extended in the cases where our results are partial (see Section 8.7). Second, and more importantly, extending this procedure to object-oriented separation logic would be fruitful to evaluate this procedure's usefulness. We say so, because, in the fragment of separation logic used in program verifiers for while languages (i.e., without the magic wand and without quantifiers), entailment checking is decidable and very fast, rendering our procedure unnecessary. In full-fledged object-oriented separation logic (i.e., including the magic wand, quantifiers, and predicates), however, entailment checking is undecidable. In this case, we believe our procedure would be useful. To extend our procedure to object-oriented separation logic, one would need to infer relations between the sizes of models of predicates and use these relations cleverly.

8.11 Related Work and Conclusion

Related Work. In the context of program verifiers, checking entailment between separation logic formulas has been studied for while languages [16] and object-oriented languages [43, 33]. In these fragments, the magic wand $-\!\!\star$ is omitted and special predicates for describing data structures are present.

Properties (decidability, undecidability, and complexity) both for model checking and for validity have been studied [30, 14, 27]. To our knowledge, disproving entailment has only been studied in a fully-fledged tableaux procedure by Galmiche et al. [48]. The cited work's algorithm for checking entailment can generate either proofs or counter-models, whereas our algorithm focuses on disproving entailment and does not generate counter-models. Galmiche et al. [48] do not provide a complexity results for their algorithm, but we believe our algorithm's complexity is much smaller.

Conclusion. We presented new techniques to disprove entailment between separation logic formulas, which we believe to be particularly relevant in the context of automated provers for separation logic. Because our algorithm's complexity is low, it is of interest wherever entailment checking needs to be checked quickly. We described two different techniques (of different precision), each of which can be applied to both the intuitionistic and classical variants of separation logic. We provided mechanical proofs of the soundness of our techniques [62].

Chapter 9

Automatic Parallelization and Optimization by Proof Rewriting

In this chapter, we present a new technique to automatically parallelize programs. We use the separating semantics of separation logic's $\star$ operator to detect when program part access separate parts of the heap. In addition to parallelization, we present various other transformations that optimize programs by taking advantage of separating informations. Transformation of programs is performed by a rewrite system.

This chapter is structured as follows: in Section 9.1 we show why automatic parallelization is a useful technique and we describe our procedure from a high-level point of view, in Section 9.2 we present the formal language that we use in this chapter (it differs from the Java-like language that is used in Chapter 2 to Chapter 7), in Section 9.3 we describe our procedure for automatic parallelization in details, in Section 9.4 we show our procedure for various optimizations, in Section 9.5 we present our implementation, in Section 9.6 we describe how our procedure would benefit of new advances of separation logic, and in Section 9.7 we discuss possible future work. Finally, we discuss related work and conclude in Section 9.8.

Note: The work from this chapter is published in the International Symposium on Static Analysis (SAS 2009) [63]. This chapter is independent of the other chapters: notations used here are unrelated with notations from other chapters.

9.1 Motivations and High Level Overview

As explained in the thesis's introduction, the high demand on software and the increasing number of processors on motherboards stimulate programmers to write multithreaded programs. This is not an easy task though: writing correct multithreaded programs is much more difficult than writing correct sequential programs.

In addition to problems we already described (data races and deadlocks), programmers have to decide *when* to parallelize. This is a difficult task, because it requires an in-depth knowledge of the data structures used. In addition, this is highly error-prone. If a program

is incorrectly parallelized, data races and deadlocks (as explained above) might happen. As the occurrence of these defects directly depends on the scheduler, they are hard to reproduce and might look random at first sight. Consequently, the relation between the incorrect parallelism and the errors might be overlooked, rendering program debugging difficult.

For the reasons explained above, automatic parallelizers [56, 49, 52] have been introduced. Automatic parallelizers take programs as input and yield parallelized programs that *should* behave the same as the input programs. Automatic parallelizers can yield programs that execute an order of magnitude faster than the input programs. One of the shortcomings of these parallelizers is that they use intricate and ad-hoc pointer analysis. As a result, showing that these parallelizers are sound is difficult.

In this chapter, we describe a new technique to infer parallelism from programs proven correct with separation logic. Instead of designing ad-hoc analysis techniques, we use separation logic to analyze programs before parallelizing them. We use separation logic's separating operator (the $*$ operator in previous chapters) – which expresses disjointness of parts of the heap – to detect potential parallelism. In a sense, we push Knuths's advice "premature optimization is the root of all evil" to an extreme: we optimize programs once they have been proven correct.

Contrary to most previous works that manipulate proofs [11, 82, 92], our algorithms manipulate proof trees representing derivations of Hoare triples. Figure 9.1 on page 153 shows the overall procedure:

(a) An external program attempts to verify the input program C.
(b) If C is verified (the verifier outputs "correct"), a proof tree is generated ($\mathcal{P}$).
(c) Then the proof tree is rewritten to obtain a parallelized and optimized program (C^{opt}) together with its proof ($\mathcal{P}^{opt}$).

Figure 9.1 indicates the step that we take for granted in white (the program verifier) and indicates our contributions in grey (the proof tree generator and the proof tree rewriter). The generation of proof trees is done with a modified version of smallfoot [16] and the rewrite system is implemented in tom [5].

9.2 Smallfoot

In this section, we present the programming language (Section 9.2.1), the variant of separation logic used to annotate programs written in this language (Section 9.2.2), and the proof rules used to verify annotated programs (Section 9.2.3). This material is not new, it has been developed by Berdine et al.'s seminal work on smallfoot [15]. We just recall the relevant parts of this work.

9.2.1 Smallfoot's Programming Language

Smallfoot's programming language is a small while language. It includes variables, expressions, simple objects, procedure, and locks:

x, y, z	$\in$	Var	variables
E, F, G	$::=$	`null` $\mid x$	expressions
b	$::=$	$E = E \mid E \neq E$	boolean expressions
$f, g, f_i, l, r, \ldots$	$\in$	FieldId	fields
p	$\in$	ProcId	procedure identifiers
l	$\in$	LockId	lock identifiers

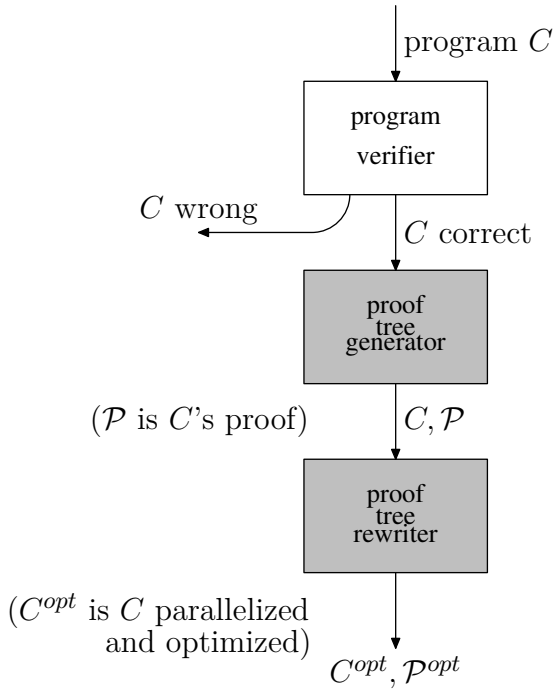

Figure 9.1: High level procedure with contributions in grey

Atomic commands A and commands C are defined by the following grammar:

$$\begin{array}{lcl} A & ::= & x := E \mid x := E \rightarrow f \mid E \rightarrow f := F \mid x := \texttt{new()} \mid \texttt{dispose}(E) \\ C & ::= & A \mid \texttt{empty} \mid \texttt{local}\ \overline{x} \mid \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}(b)\{C\} \\ & \mid & \texttt{lock}(l) \mid \texttt{unlock}(l) \mid p(\overline{E_1}; \overline{E_2}) \mid C; C \mid C \| C' \end{array}$$

The meaning of atomic commands is as follows:

- Command $x := E$ assigns E to variable x.
- Command $x := E \rightarrow f$ looks up the content of field f of the cell at address E and stores $E \rightarrow f$'s content in variable x.
- Command $E \rightarrow f := F$ assigns E to field f of the cell at address E.
- Command $x := \texttt{new()}$ allocates a new cell and assigns the new cell's address to x.
- Command $\texttt{dispose}(E)$ disposes the cell pointed to by E.

In $\texttt{lock}(l)$ and $\texttt{unlock}(l)$[1], l is a non-reentrant lock [84]. Locks are declared in the (omitted) program's header and — like in Chapter 5 — come with a *resource invariant*, i.e., a formula describing the part of the heap guarded by locks. Intuitively, when a lock is acquired by a process, the lock's resource invariant is transferred to the process; while when a lock is released, the lock's resource invariant is transferred from the process back to the lock. In procedure calls $p(\overline{E_1}; \overline{E_2})$, $\overline{E_1}$ are the parameters that are unchanged in p's body, while $\overline{E_2}$ are the parameters that may be assigned to in p's body.

The operational semantics of the language is omitted, because it is standard [15].

9.2.2 Smallfoot's Assertion Language

Programs written in smallfoot's language should be annotated with separation logic contracts. Here, we present the variant of separation logic used to write contracts.

Our assertion language distinguishes between pure (heap independent) and spatial (heap dependent) assertions:

$$\begin{array}{rcll} \Pi & ::= & \texttt{true} \mid b \mid \Pi \wedge \Pi & \text{pure formulas} \\ \rho & ::= & f_1 : E_1, \ldots, f_n : E_n & \text{record expressions} \\ S & ::= & E \mapsto [\rho] \mid \texttt{list}(E) \mid \texttt{tree}(E) & \text{simple spatial formulas} \\ \Sigma & ::= & \texttt{emp} \mid S \mid \Sigma \star \Sigma & \text{spatial formulas} \\ \Xi, \Theta & \in & \Pi \mid \Sigma & \text{formulas} \end{array}$$

The meaning of the simple spatial formulas is as follows: $\texttt{emp}$ represents the empty heap, $E \mapsto [\rho]$ represents a heap containing one cell at address E with content ρ, $\texttt{list}(E)$ represents a heap containing a list at address E, and $\texttt{tree}(E)$ represents a heap containing a tree whose root is at address E and whose left and right subtrees can be dereferenced with fields l and r. The formula $E \mapsto [\rho]$ can mention any number of fields in ρ: the values of omitted fields are implicitly existentially quantified. Top-level formulas are pairs $\Pi \mid \Sigma$ where Π is a $\wedge$-separated sequence of pure formulas (indicating equalities/inequalities between expressions) and Σ is a $\star$-separated sequence of spatial formulas (indicating facts about the heap).

[1] To smallfoot's experts: smallfoot uses conditional critical regions `with do endwith`, instead of `lock`/`unlock` commands. However, because smallfoot generates *verification conditions* [16], conditional critical regions are treated like `lock`/`unlock` commands in smallfoot's implementation. That is why we use `lock`/`unlock` commands in our presentation.

The $\star$ operator (like $*$ in previous chapters) is the separation conjunction. Intuitively, if formula $\Pi \mid (\Sigma \star \Sigma')$ holds, this means that the heap can be split in two *disjoint* subheaps such that $\Pi \mid \Sigma$ represents the first subheap and $\Pi \mid \Sigma'$ represents the second subheap.

We define the $\wedge$-conjunction of a pure formula and a formula:

$$\begin{array}{rcl} \Pi \wedge (\Pi' \mid \Sigma) & \triangleq & (\Pi \wedge \Pi') \mid \Sigma \\ (\Pi \mid \Sigma) \wedge \Pi' & \triangleq & (\Pi \wedge \Pi') \mid \Sigma \end{array}$$

Similarly, we define the $\star$-conjunction of a spatial formula and a formula:

$$\begin{array}{rcl} \Sigma \star (\Pi \mid \Sigma') & \triangleq & \Pi \mid (\Sigma \star \Sigma') \\ (\Pi \mid \Sigma) \star \Sigma' & \triangleq & \Pi \mid (\Sigma \star \Sigma') \end{array}$$

Finally, we define the $\star$-conjunction of two formulas:

$$(\Pi \mid \Sigma) \star (\Pi' \mid \Sigma') \quad \triangleq \quad (\Pi \wedge \Pi') \mid (\Sigma \star \Sigma')$$

Entailment between formulas is written $\Xi \vdash \Theta$. We lift $\vdash$ to pure formulas as follows:

$$\Pi \vdash \Pi' \quad \triangleq \quad \Pi \mid \mathtt{emp} \vdash \Pi' \mid \mathtt{emp}$$

We use σ to range over substitutions of the form $x_0/y_0, \ldots, x_n/y_n$. Below we abusively write $\Pi \vdash x_0/y_0, \ldots, x_n/y_n$ to denote $\Pi \vdash x_0 = y_0 \wedge \cdots \wedge x_n = y_n$. We define a syntactical equivalence relation between formulas as follows:

$$\Pi \mid \Sigma \Leftrightarrow \Pi' \mid \Sigma' \text{ iff } \begin{cases} \Pi \text{ is a permutation of } \Pi' \\ \exists \sigma, \Pi \vdash \sigma \text{ and } \Sigma[\sigma] \text{ is a permutation of } \Sigma' \end{cases}$$

The semantics of formulas is omitted and can be found in Berdine et al.'s work [16]. Importantly, the semantics is *classical* (contrary to the semantics used from Chapter 3 to Chapter 7). Classical separation logic is suitable to reason about programming languages where programmers have to deallocate memory manually. The programming language we use is such a language, because it has a `dispose` statement.

9.2.3 Smallfoot's Proof Rules

Smallfoot programs are verified using a set of Hoare rules. Here, we only show the Hoare rules[2] for atomic commands because they are important for the algorithms we develop later (see Section 9.3).

In these rules, we use the following notations to mutate and lookup the content of records:

$$\mathsf{mutate}(\rho, f, F) = \begin{cases} f : F, \rho' & \text{if} \quad \rho = f : E, \rho' \\ f : F, \rho & \text{if} \quad f \notin \rho \end{cases} \qquad \mathsf{lkp}(\rho, f) = \begin{cases} E & \text{if} \quad \rho = f : E, \rho' \\ x \text{ fresh} & \text{if} \quad f \notin \rho \end{cases}$$

The first Hoare rule describes the effect of variable assignment:

[2] To smallfoot's experts: in the original paper on smallfoot, the Hoare rules are presented in an algorithmic style with continuations. We do not follow this presentation, because it does not fit well with our explanations. We use a rule for sequential composition instead of including continuations in all rules. In addition, we use standard Hoare rules for while loops, parallel composition, etc., whereas smallfoot uses "subroutines". The resulting verification systems are, anyway, equivalent: showing a program correct with the proof rules we describe here is equivalent to showing this program correct with smallfoot's original rules.

$$\frac{x' \text{ is fresh}}{\{\Pi \mid \Sigma\} x := E \{x = E[x'/x] \wedge (\Pi \mid \Sigma[x'/x])\}} \text{ (Assign)}$$

In rule (Assign), the fresh variable x' is used to record x's value before the assignment. In (Assign)'s postcondition, once the substitution x'/x has been applied, the unique x remaining (in $x = E[x'/x]$) keeps track of the "old" value. This standard trick is also used in other rules.

The following rule verifies heap lookup[3]:

$$\frac{\Pi \vdash F = E \qquad \mathsf{lkp}(\rho, f) = G \qquad x' \text{ is fresh}}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} x := E \rightarrow f \{x = G[x'/x] \wedge (\Pi \mid \Sigma \star F \mapsto [\rho])[x'/x]\}} \text{ (Lookup)}$$

The rule (Lookup) requires that the dereferenced cell (pointed to by E) is allocated. This follows from $F \mapsto [\rho]$ (which appears in the precondition) and $F = E$. By combining these two assertions, one can derive $E \mapsto [\rho]$, i.e., the cell pointed to by E is allocated.

The next rule verifies mutations to the heap:

$$\frac{\Pi \vdash F = E \qquad \mathsf{mutate}(\rho, f, G) = \rho'}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} E \rightarrow f := G \{\Pi \mid \Sigma \star F \mapsto [\rho']\}} \text{ (Mutate)}$$

The rule (Mutate) requires that the dereferenced cell (pointed to by E) is allocated. Like in the rule (Lookup), this follows from $F \mapsto [\rho]$ and $F = E$. Rule (Mutate)'s postcondition reflects the change to the heap performed by the command.

Finally, there are two rules for allocation and deallocation:

$$\frac{x' \text{ is fresh}}{\{\Pi \mid \Sigma\} x := \texttt{new()} \{(\Pi \mid \Sigma)[x'/x] \star x \mapsto []\}} \text{ (New)}$$

$$\frac{\Pi \vdash F = E}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} \texttt{dispose(}E\texttt{)} \{\Pi \mid \Sigma\}} \text{ (Dispose)}$$

The rule (New) verifies that a new cell is allocated: $x \mapsto []$ appears in the postcondition. Rule (Dispose) reflects that a cell is disposed: $F \mapsto [\rho]$ appears in the precondition, but not in the postcondition.

We do not show the rules for conditionals, loops, locks, and method calls, as they are standard. We show, however, three rules that are at the basis of further developments: a rule for parallel composition, a frame rule, and a rule for sequential composition.

First, we show a rule for parallel composition [84]:

$$\frac{\{\Xi\} C \{\Xi'\} \qquad \{\Theta\} C' \{\Theta'\}}{\{\Xi \star \Theta\} C \| C' \{\Xi' \star \Theta'\}} \text{ (Parallel)}$$

Side condition: C does not modify any variables free in Θ, C', and Θ' (and conversely).

The rule (Parallel) can be understood as follows: for two threads to execute in parallel, they should access disjoint parts of the heap (i.e., Ξ and Θ). When the two threads terminate, the resulting heap is simply the $\star$ conjunction of the heaps obtained after both threads's execution (i.e., Ξ' and Θ').

Second, smallfoot's verification system admits a (Frame) rule [93, 83]. This means that the following rule can be added to smallfoot's set of proof rules without breaking soundness:

[3]Where we include Berdine et al.'s "rearrangement" step. This means that the postcondition can mention a variable (F) different from the dereferenced program's variable (E) as long as $F = E$ holds.

$$\frac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi_f\}C\{\Theta \star \Xi_f\}}\ (\text{Frame } \Xi_f)$$

Side condition: C does not modify any variables in Ξ_f

The rule (Frame) allows reasoning about a program in isolation from its environment, by focusing only on the part of the heap that this program accesses: This is expressed as follows: if C's execution safely transforms heap Ξ into heap Θ (i.e., $\{\Xi\}C\{\Theta\}$ holds), then C's execution safely transforms the "bigger" heap $\Xi \star \Xi_f$ into the heap $\Theta \star \Xi_f$ (i.e., $\{\Xi \star \Xi_f\}C\{\Theta \star \Xi_f\}$ holds). The (Frame) rule is the essence of local reasoning. This means that one should prove "small" triples ($\{\Xi\}C\{\Theta\}$) and then use such triples in bigger contexts ($\{\Xi \star \Xi_f\}C\{\Theta \star \Xi_f\}$).

In the (Frame) above, formula Ξ_f is called the frame, it is the part of the heap that is unaffected by C's execution; while formula Ξ is called the *antiframe* [29], it is the part of the heap that is affected by C's execution.

Third, smallfoot admits a rule for sequential composition. This rule is not present in the seminal work on smallfoot, because this work focuses on algorithmic checking, but it is admissible:

$$\frac{\{\Xi\}C\{\Xi'\} \qquad \{\Xi'\}C\{\Theta\}}{\{\Xi\}C;\, C'\{\Theta\}}\ (\text{Seq})$$

9.3 Automatic Parallelization

In this section, we describe our algorithm for parallelizing programs proven correct. As sketched in this chapter's introduction, our algorithm takes programs (with their proofs) as input and yields (parallelized) programs as output (with their proofs). Proofs are a derivation of Hoare triples and our algorithm is a rewrite system on such derivations.

9.3.1 High-Level Procedure

Our algorithm for rewriting proof trees focuses on two rules of separation logic: the (Frame) rule and the (Parallel) rule. The basic idea of our reasoning is depicted by the following rewrite rule:

$$\frac{\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\ (\text{Frame } \Xi') \qquad \dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\ (\text{Frame } \Theta)}{\{\Xi \star \Xi'\}C;\, C'\{\Theta \star \Theta'\}}\ (\text{Seq})}{\downarrow \underline{\text{Parallelize}}^{4}}$$

$$\frac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \| C'\{\Theta \star \Theta'\}}\ (\text{Parallel})$$

The diagram above should be read as follows: Given a proof of the sequential program $C;\, C'$ we rewrite this proof into a proof of the parallel program $C \| C'$. If the initial proof tree is valid, this rewriting yields a valid proof tree because the leaves of the rewritten proof tree are included in the leaves of the initial proof tree.

Intuitively, parallelizing C and C' is sound because the left-hand side of rule Parallelize makes sure that command C does not access the part of the heap accessed by C': the

[4] To disambiguate between Hoare rules (enclosed in parentheses) and rewrite rules, we underline rewrite rules.

antiframe of C' (Ξ') is framed to prove that executing C is safe (see (Frame Ξ')). Dually, command C' does not access the antiframe of c (Ξ). For these two reasons, it is safe to execute C and C' in parallel.

Formally, rewrite rules are relations $\mathcal{P} \rightarrow \mathcal{P}'$ that take an input proof tree $\mathcal{P}$ and yield an output proof tree $\mathcal{P}'$. A rewrite rule $\rightarrow$ is *sound* iff for all valid proof trees $\mathcal{P}$ such that $\mathcal{P} \rightarrow \mathcal{P}'$, $\mathcal{P}'$ is valid. A proof tree is *valid* if each inference is an instance of a proof rule.

The rewrite rules we present in this chapter satisfy the following properties: (1) the rewrite rules are sound and (2) the rewrite rules preserve specifications i.e., given a proof tree whose root is $\{\Xi\}\_\{\Theta\}$, any tree returned by the rewrite system (consisting of all rewrite rules from this chapter) will have $\{\Xi\}\_\{\Theta\}$ as its root. This holds simply because all our rewrite rules leave the pre/postcondition of the root of the input proof tree untouched.

We conjecture that our rewrite system actually provides a stronger guarantee than preserving specifications. Plausibly, the set of final states of an input program and the set of final states of the corresponding optimized program are related. We leave this study, however, as future work.

9.3.2 Proof Rules with Explicit Antiframes and Frames

In this section we show why smallfoot's proof rules should be modified for our algorithm to work properly. We provide appropriate proof rules.

As Parallelize's left-hand side shows, the (Frame) rule is the central ingredient of our procedure: for Parallelize to fire, we need applications of (Frame) inside proof trees. In practice, however, frames are not systemically computed at atomic commands. To exemplify this statement, consider smallfoot's proof rule for heap lookup (Lookup):

$$\frac{\mathsf{lkp}(\rho, f) = G \qquad \Pi \vdash F = E \qquad x' \text{ is fresh}}{\{\Pi \mid \Sigma \star F \mapsto [\rho]\} x := E \rightarrow f \{x = G[x'/x] \wedge (\Pi \mid \Sigma \star F \mapsto [\rho])[x'/x]\}} \text{ (Lookup)}$$

The rule (Lookup) does not frame the precondition: the whole pure part of the precondition (Π) is used to show $F = E$ and the substitution x'/x is applied to the whole precondition ($\Pi \mid \Sigma \star F \mapsto [\rho]$). In other words, this rule does make explicit the part of the precondition that is framed i.e., (1) the pure part of the precondition that is useless to show $F = E$ and (2) the part of the precondition that is left unaffected by the substitution x'/x.

A similar remark applies to the other rules for atomic commands: (Assign),(Mutate), (New), and (Dispose). To remedy this problem, we propose new proof rules for atomic commands where antiframes and frames are made explicit. Figure 9.2 shows rules (derived from smallfoot's proof rules) for each atomic command where antiframes and frames are made explicit. In these rules, we subscript formulas representing antiframes by a and formulas representing frames by f. Extra side conditions on the applications of (Frame) are indicated as additional premises.

To help the reader understand these rules, we detail the rule exhibiting the antiframe and frame at a field lookup command (the second rule). The antiframe $\Pi_a \mid \Sigma_a \star F \mapsto [\rho]$ consists of (1) the pure part of the precondition which is necessary to show $F = E$: it is Π_a and of (2) the spatial part of the precondition asserting that the cell at E exists

$$\dfrac{\dfrac{x' \text{ fresh}}{\{\Xi_a\}x := E\{\Xi_a[x'/x]\}}\,(\text{Assign}) \qquad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := E\{\Xi_a[x'/x] \star \Xi_f\}}\,(\text{Frame } \Xi_f)$$

$$\dfrac{\dfrac{\begin{array}{c}\Pi_a \vdash F = E \quad x' \text{ fresh} \quad \mathsf{lkp}(\rho, f) = G \\ \Xi = \Pi_a[x'/x] \wedge x = G[x'/x] \mid (\Sigma_a \star F \mapsto [\rho])[x'/x]\end{array}}{\{\Pi_a \mid \Sigma_a \star F \mapsto [\rho]\}x := E \to f\{\Xi\}}\,(\text{Lookup}) \qquad x \notin \Xi_f}{\{(\Pi_a \mid \Sigma_a \star F \mapsto [\rho]) \star \Xi_f\}x := E \to f\{\Xi \star \Xi_f\}}\,(\text{Frame } \Xi_f)$$

$$\dfrac{\dfrac{\Pi_a \vdash F = E \quad \mathsf{mutate}(\rho, f, G) = \rho'}{\{\Pi_a \mid F \mapsto [\rho]\}E \to f := G\{\Pi_a \mid F \mapsto [\rho']\}}\,(\text{Mutate})}{\{\Pi_a \mid F \mapsto [\rho] \star \Xi_f\}E \to f := G\{\Pi_a \mid F \mapsto [\rho'] \star \Xi_f\}}\,(\text{Frame } \Xi_f)$$

$$\dfrac{\dfrac{x' \text{ fresh}}{\{\Xi_a\}x := \texttt{new()}\{\Xi_a[x'/x] \star x \mapsto []\}}\,(\text{New}) \qquad x \notin \Xi_f}{\{\Xi_a \star \Xi_f\}x := \texttt{new()}\{\Xi_a[x'/x] \star x \mapsto [] \star \Xi_f\}}\,(\text{Frame } \Xi_f)$$

$$\dfrac{\dfrac{\Pi_a \vdash F = E}{\{\Pi_a \mid F \mapsto [\rho]\}\texttt{dispose(}E\texttt{)}\{\Pi_a \mid \texttt{emp}\}}\,(\text{Dispose})}{\{\Pi_a \mid F \mapsto [\rho] \star \Xi_f\}\texttt{dispose(}E\texttt{)}\{\Pi_a \mid \texttt{emp} \star \Xi_f\}}\,(\text{Frame } \Xi_f)$$

Figure 9.2: Derived rules for atomic commands with explicit antiframes and frames

$(F \mapsto [\rho])$ and the spatial part of the precondition affected by the substitution x'/x (Σ_a). The frame is the antiframe's complement (Ξ_f).

Theorem 14. *The rules in Figure 9.2 are sound.*

Proof. Observe that the restrictions imposed on explicit frames ($x \notin \Xi_f$) make explicit frames immune to substitutions x'/x (cases (Assign), (Lookup), and (New)). Then, further observe that these rules derive from [16]'s rules (which are sound). □

We have not discussed the rule for method calls. That is intentional: existing proof rules for method calls [93, 15] already compute frames and antiframes at procedure calls. Similarly, smallfoot makes frames and antiframes explicit at lock and unlock primitives. Finally, we use standard rules for while loops and conditionals. There is a caveat though: because smallfoot generates verification conditions [15], proofs for while loops are "separated" from the enclosing method. This will forbid to move code from within a loop outside of the loop (and vice versa).

9.3.3 Frames Factorization

In this section, we show that explicit framing is not yet sufficient to obtain "good" frames. We give a solution to this problem: we factorize frames.

Most proof trees generated by Figures 9.2's rules do not match the left-hand side of the rewrite rule Parallelize shown on page 157. To exemplify this statement, we define the following abbreviation:

$$\dfrac{\dfrac{\dfrac{}{\{\Lambda^{\_}_x\}x\to f := E\{\Lambda^E_x\}}\,(\text{Mutate})}{\{\Lambda^{\_,\_,\_}_{x,y,z}\}x\to f := E\{\Lambda^{E,\_,\_}_{x,y,z}\}}\,(\text{Fr }\Lambda^{\_,\_}_{y,z}) \quad \dfrac{\dfrac{}{\{\Lambda^{\_}_y\}y\to f := F\{\Lambda^F_y\}}\,(\text{Mutate})}{\{\Lambda^{E,\_,\_}_{x,y,z}\}y\to f := F\{\Lambda^{E,F,\_}_{x,y,z}\}}\,(\text{Fr }\Lambda^{E,\_}_{x,z}) \quad \dfrac{\dfrac{}{\{\Lambda^{\_}_z\}z\to f := G\{\Lambda^G_z\}}\,(\text{Mutate})}{\{\Lambda^{E,F,\_}_{x,y,z}\}z\to f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Fr }\Lambda^{E,F}_{x,y})}{\{\Lambda^{E,\_,\_}_{x,y,z}\}y\to f := F;\ z\to f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Seq})}\,(\text{Seq})$$
$$\{\Lambda^{\_,\_,\_}_{x,y,z}\}x\to f := E;\ y\to f := F;\ z\to f := G\{\Lambda^{E,F,G}_{x,y,z}\}$$

Figure 9.3: A proof tree obtained by applying Figure 9.2's rules

$$\Lambda^{E_0,\ldots,E_m}_{x_0,\ldots,x_m} \triangleq x_0\mapsto[f : E_0]\star\cdots\star x_m\mapsto[f : E_m]$$

Note that this abbreviation enjoys the following equivalence:

$$\Lambda^{E_0,\ldots,E_m,E_{m+1},\ldots,E_{m+k}}_{x_0,\ldots,x_m,x_{m+1},\ldots,x_{m+k}} \Leftrightarrow \Lambda^{E_0,\ldots,E_m}_{x_0,\ldots,x_m}\star\Lambda^{E_{m+1},\ldots,E_{m+k}}_{x_{m+1},\ldots,x_{m+k}}$$

Now, to see why proof trees generated by Figure 9.2's rules do not match the left-hand side of the rewrite rule Parallelize, consider the proof tree shown in Figure 9.3 (where pure formulas are omitted, (Fr) abbreviates (Frame), and _ denotes existentially quantified values). The rewrite rule Parallelize cannot fire on Figure 9.3's proof tree because this proof tree contains applications of (Frame) at each atomic command. Generally, given a program $A_0; A_1; \ldots$, the proof rules with explicit frames generate a proof tree with the following shape:

$$\dfrac{\dfrac{\ldots}{\{\ldots\}A_0\{\ldots\}}\,(\text{Frame}) \quad \dfrac{\dfrac{\ldots}{\{\ldots\}A_1\{\ldots\}}\,(\text{Frame}) \quad \ldots}{\{\ldots\}A_1;\ldots\{\ldots\}}\,(\text{Seq})}{\{\ldots\}A_0;\ A_1;\ldots\{\ldots\}}\,(\text{Seq})$$

Proof trees with the shape above are inappropriate for the rewrite rule Parallelize. Intuitively, the problem lies in the successive applications of (Frame) being redundant: the same formula is framed multiple times. For example, in the proof tree shown in Figure 9.3, the formula Λ^E_x is framed twice: once in the center (Frame) and once in the right (Frame).

More generally, the presence of redundant frames means that applications of (Frame) are on *short* commands (i.e., atomic commands or a small sequence of commands). However, as the left-hand side of the rewrite rule for parallelization described in the introduction shows, to parallelize *long* commands (i.e., long sequence of commands), applications of (Frame) have to be on long commands. Hence, removing redundant frames is a mandatory step before parallelizing. We solve this issue by inferring applications of (Frame) on long commands from application of (Frame) on short commands.

The redundancy in applications of (Frame) originally arises from smallfoot's symbolic execution algorithm, that implements the set of Hoare rules shown in Section 9.2.3. Because symbolic execution mimics an operational update of the state at each atomic command, each atomic command is treated independently. To fix this issue, two solutions are possible. The first solution is to build a new program verifier that infers frames for non-atomic commands. We think this solution is inadequate because it requires to design a program verifier with proof rewriting in mind (breaking separation of concerns). The second solution, that we choose here, is to minimize the modifications of the program verifier and to do as much work as possible on the proof rewriting side.

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,(\text{Frame } \Xi_f) \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Theta_f) \qquad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_f\}C';\, C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, C';\, C''\{\Xi'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{FactorizeFrames}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f_0}\}C\{\Xi_p \star \Xi_{f_0}\}}\,(\text{Frame } \Xi_{f_0}) \qquad \dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{f_0}\}C'\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Frame } \Theta_{f_0})}{\{\Xi_a \star \Xi_{f_0}\}C;\, C'\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\, C'\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Xi_c) \qquad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Xi_a \star \Xi_f\}C;\, C';\, C''\{\Xi'\}}\,(\text{Seq})$$

Guard: $\Xi_f \Leftrightarrow \Xi_{f_0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f_0} \star \Xi_c$

Figure 9.4: Rewrite rule to factorize applications of (Frame)

$$\dfrac{\dfrac{\{\Lambda^-_x\}x \rightarrow f := E\{\Lambda^E_x\}}{\{\Lambda^{-,-,-}_{x,y,z}\}x \rightarrow f := E\{\Lambda^{E,-,-}_{x,y,z}\}}\,(\text{Fr } \Lambda^{-,-}_{y,z}) \qquad \dfrac{\dfrac{\dfrac{}{\{\Lambda^-_y\}y \rightarrow f := F\{\Lambda^F_y\}}\,(\text{Mutate})}{\{\Lambda^{-,-}_{y,z}\}y \rightarrow f := F\{\Lambda^{F,-}_{y,z}\}}\,(\text{Fr } \Lambda^-_z) \qquad \dfrac{\dfrac{}{\{\Lambda^-_z\}z \rightarrow f := G\{\Lambda^G_z\}}\,(\text{Mutate})}{\{\Lambda^{F,-}_{y,z}\}z \rightarrow f := G\{\Lambda^{F,G}_{y,z}\}}\,(\text{Fr } \Lambda^F_y)}{\{\Lambda^{-,-}_{y,z}\}y \rightarrow f := F;\, z \rightarrow f := G\{\Lambda^{F,G}_{y,z}\}}\,(\text{Seq})}{\{\Lambda^{E,-,-}_{x,y,z}\}y \rightarrow f := F;\, z \rightarrow f := G\{\Lambda^{E,F,G}_{x,y,z}\}}\,(\text{Fr } \Lambda^E_x)$$

$$\{\Lambda^{-,-,-}_{x,y,z}\}x \rightarrow f := E;\, y \rightarrow f := F;\, z \rightarrow f := G\{\Lambda^{E,F,G}_{x,y,z}\}\,(\text{Seq})$$

Figure 9.5: Figure 9.3's proof tree after applying FactorizeFrames once

Figure 9.4 shows the rewrite rule FactorizeFrames that removes redundancy in applications of (Frame). FactorizeFrames fires if C and C' are two consecutive commands that both frame a part of the state (Ξ_f and Θ_f respectively) such that the two parts of the state share a common part (Ξ_c as imposed by the guard). In FactorizeFrames's right-hand side, the common part of the state is framed *once*, below the application of (Seq).

Both the left-hand side of FactorizeFrames and the right-hand side of FactorizeFrames include the proof tree of the Hoare triple $\{\Theta_p \star \Theta_f\}C''\{\Xi'\}$. We need to include such a proof tree to match two possible cases: C'' can be a dummy "continuation" (represented by the `empty` command) or a "normal" continuation. In the implementation, all rewrite rules use this "possible continuation" trick. For clarity reasons, however, in the rest of this chapter, we describe rewrite rules without such continuations and refer the interested reader to Appendix C for full rules.

Figure 9.5 exemplifies an application of FactorizeFrames to Figure 9.3's proof tree: the redundancy of Λ^E_x in the center and the right (Frame)s is factorized in a single (Frame).

Theorem 15. *The rewrite rule FactorizeFrames is sound.*

Proof. Suppose the left-hand side of FactorizeFrames is valid. The goal is to show that the right-hand side of FactorizeFrames is valid.

For the application of (Frame Ξ_c) to be valid, we must show the two following equivalences: $\Xi_a \star \Xi_f \Leftrightarrow \Xi_a \star \Xi_{f_0} \star \Xi_c$ and $\Theta_p \star \Theta_f \Leftrightarrow \Theta_p \star \Theta_{f_0} \star \Xi_c$. But these two equivalences

follow directly from FactorizeFrames's guard.

For the application of (Frame Θ_{f_0}) to be valid, we must show the following equivalence:

$$\Xi_p \star \Xi_{f_0} \Leftrightarrow \Theta_a \star \Theta_{f_0} \tag{goal}$$

From FactorizeFrames's first guard, we obtain:

$$\Xi_p \star \Xi_f \Leftrightarrow \Xi_p \star \Xi_{f_0} \star \Xi_c \tag{equiv 1}$$

From the validity of the application of (Seq) in FactorizeFrames's left-hand side, we obtain: $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_f$. Then, from FactorizeFrames's second guard, we obtain:

$$\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_{f_0} \star \Xi_c \tag{equiv 2}$$

By transitivity and symmetry of $\Leftrightarrow$, equiv 1, and equiv 2, we obtain:

$$\Xi_p \star \Xi_{f_0} \star \Xi_c \Leftrightarrow \Theta_a \star \Theta_{f_0} \star \Xi_c \tag{9.1}$$

By cancelling Ξ_c on the right hand sides of the last equivalence, we obtain the desired goal. Now FactorizeFrames's validity is deduced as follows: (1) each inference in FactorizeFrames's right-hand side is a valid instance of the proof rules and (2) the leaves of FactorizeFrames's right-hand side are identical to the leaves of FactorizeFrames's left-hand side (which are valid by hypothesis). □

9.3.4 Parallelization in Practice

In practice, factorizing frames is a mandatory step before applying the Parallelize rewrite rule shown in the introduction. For example, applying Parallelize to the proof tree shown in Figure 9.5 yields a proof of the following Hoare triple:

$$\{\Lambda^{\_,\_,\_}_{x,y,z}\} x \rightarrow f := E \parallel (y \rightarrow f := F \parallel z \rightarrow f := G) \{\Lambda^{E,F,G}_{x,y,z}\}$$

For the Parallelize rewrite rule to be sound, we add the guard that C does not modify variables in Ξ', C', and Θ' (and conversely). Note that, for clarity of presentation, the Parallelize rule shown in Section 9.3.1 does not include the "possible continuation" trick mentioned above. We refer the interested reader to Appendix C for the full rule.

Theorem 16. *The rewrite rule Parallelize is sound.*

Proof. Suppose Parallelize's left-hand side is valid. The goal is to show that Parallelize's right-hand side is valid.

This holds for the two following reasons: (1) Parallelize's right-hand side is a valid application of (Parallel) (because the guard is exactly (Parallel)'s side condition) and (2) the leaves of Parallelize's right-hand side are identical to the leaves of Parallelize's left-hand side (which are valid by hypothesis). □

9.3.5 Examples of Parallelization

Figure 9.6 shows procedure `disp_tree` (borrowed from smallfoot's example suite) that takes a tree at x and recursively disposes it (hence the postcondition `emp`). If the tree is `null` then `disp_tree` simply returns. Otherwise, the tree's leaves are looked up, `disp_tree` is called on the leaves, and the root of the tree is disposed. Applying

```
requires tree(x);
ensures  emp;
disp_tree(x;){
 local i, j;
 if(x = null){}
 else{
  i := x→l; j := x→r;
  disp_tree(i;);
  disp_tree(j;);
  dispose(x);
 }
}
```

→

```
requires tree(x);
ensures  emp;
disp_tree(x;){
 local i, j;
 if(x = null){}
 else{
  i := x→l; j := x→r;
  disp_tree(i;) ||
  disp_tree(j;) ||
  dispose(x);
 }
}
```

Figure 9.6: Parallelization of tree disposal

FactorizeFrames and Parallelize to `disp_tree` yields a program where the recursive calls to `disp_tree` and the disposal of the root are executed in parallel.

Figure 9.7 shows procedure `rotate_tree` (borrowed from Raza et al. [92]) that takes a tree at x and rotates it by recursively swapping its left and right subtrees. Applying FactorizeFrames and Parallelize to `rotate_tree` yields a program where the field assignments and the recursive calls are executed in parallel. We achieve better parallelism than Raza et al. [92] where only the recursive calls are parallelized.

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
 local x1, x2;
 if(x = null){}
 else{
  x1 := x→l;
  x2 := x→r;
  x→l := x2;
  x→r := x1;
  rotate_tree(x1;);
  rotate_tree(x2;);
 }
}
```

→

```
requires tree(x);
ensures  tree(x);
rotate_tree(x;){
 local x1, x2;
 if(x = null){}
 else{
  x1 := x→l;
  x2 := x→r;
  (x→l := x2; x→r := x1) ||
  rotate_tree(x1;) ||
  rotate_tree(x2;);
 }
}
```

Figure 9.7: Parallelization of tree rotation

9.4 Automatic Optimization

In this section, we show a generic optimization that changes the program's order (Section 9.4.1) and we present a rewrite rule that optimizes temporal locality [96] (Section 9.4.2).

9.4.1 Generic Optimization

We present an optimization that changes the program's execution order. This optimization has four concrete applications: (1) dispose memory as soon as possible to avoid out of memory errors, (2) allocate memory as late as possible to leave more allocatable memory, (3) release locks as soon as possible to increase parallelism, and (4) acquire locks as late as possible to increase parallelism. Figure 9.8 shows the rewrite rule GenericOptimization that changes the program's execution order.

GenericOptimization fires if the program has the shape $C; C'; C''$ such that C' frames the postcondition of C (as imposed by the guard). Then, the program's order is changed so that C' executes before C. It should be noted that this rule imposes that C frames the precondition of C' by the following reasoning: for the first application of (Seq) to be valid in GenericOptimization's left-hand side, we have $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Theta_f$. From the guard, it follows that $\Xi_p \star \Xi_f \Leftrightarrow \Theta_a \star \Xi_p \star \Xi_r$. By simplifying Ξ_p on both sides, we obtain: $\Xi_f \Leftrightarrow \Theta_a \star \Xi_r$. We can conclude that C frames the precondition of C' (Θ_a).

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,(\text{Fr } \Xi_f) \quad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\,(\text{Fr } \Theta_f) \quad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_f\}C'; C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C; C'; C''\{\Xi'\}}\,(\text{Seq})}{\downarrow \text{GenericOptimization}}$$

$$\dfrac{\dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Xi_a \star \Theta_a \star \Xi_r\}C'\{\Xi_a \star \Theta_p \star \Xi_r\}}\,(\text{Fr } \Xi_r \star \Xi_a) \quad \dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Theta_p \star \Xi_r\}C\{\Xi_p \star \Theta_p \star \Xi_r\}}\,(\text{Fr } \Theta_p \star \Xi_r) \quad \{\Xi_p \star \Theta_p \star \Xi_r\}C''\{\Xi'\}}{\{\Xi_a \star \Theta_p \star \Xi_r\}C; C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C'; C; C''\{\Xi'\}}\,(\text{Seq})}{\text{Guard: } \Theta_f \Leftrightarrow \Xi_p \star \Xi_r}$$

Figure 9.8: Rewrite rule to change the program's execution order

We now detail GenericOptimization's four concrete applications. (1) If C' is a `dispose` command and C frames the precondition of C', it means that C does not access the state disposed by C': better execute C' first to dispose memory as soon as possible. (2) If C is a `new` command and C' frames the postcondition of C, it means that C' does not access the state allocated by C: better execute C after C' to leave C' more allocatable memory. (3) If C' is an `unlock` command and C frames the precondition of C' (i.e. the lock's resource invariant), it means that C does not access the part of the heap represented by the lock's resource invariant: better execute C' to release the lock first. (4) If C is a `lock` command and C' frames the postcondition of C (i.e. the lock's resource invariant), it means that C' does not access the part of the heap represented by the lock's resource invariant: better execute C after C' to acquire the lock as late as possible.

Theorem 17. *The rewrite rule GenericOptimization is sound.*

Proof. Apply the guard's equivalence in the right places and observe that (1) each inference in GenericOptimization's right-hand side is a valid instance of the proof rules and (2) the leaves of GenericOptimization's right-hand side are identical to the leaves of GenericOptimization's left-hand side (which are valid by hypothesis). □

9.4.2 Optimization of Temporal Locality

Temporal locality [96] is the time between successive accesses to a heap cell. The lower this time is, the faster a program can execute because the underlying processors are less likely to access the main memory. Ideally, if a heap cell is accessed by two instructions successively, this allows the executing processor to keep the content of this heap cell in the cache, hence reducing execution time (no load/store to the main memory).

Because our framework does not express properties at the level of memory, we cannot formally speak about temporal locality. However, a rewrite rule that optimizes temporal locality can be written at our level of abstraction: heaps denoted by separation logic formulas.

Intuitively, a program $C; C'; C''$ such that C and C'' access a part of the heap disjoint from the part of the heap accessed by C' can be transformed into $C; C''; C'$ to improve temporal locality. In the optimized program, C and C'' execute consecutively: because C and C'' access the same part of the heap, the temporal locality of the optimized program is improved (i.e., it is reduced). Figure 9.11 presents our rewrite rule for improving temporal locality.

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}}{\{\Xi \star \Theta\}C\{\Xi' \star \Theta\}}\,(\text{Fr }\Theta) \qquad \dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi' \star \Theta\}C'\{\Xi' \star \Theta'\}}\,(\text{Fr }\Xi')}{\{\Xi \star \Theta\}C;\, C'\{\Xi' \star \Theta'\}}\,(\text{Seq}) \qquad \dfrac{\{\Xi'\}C''\{\Xi''\}}{\{\Xi' \star \Theta'\}C''\{\Xi'' \star \Theta'\}}\,(\text{Fr }\Theta')}{\{\Xi \star \Theta\}C;\, C';\, C''\{\Xi'' \star \Theta'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{TemporalLocality}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\} \qquad \{\Xi'\}C''\{\Xi''\}}{\{\Xi\}C;\, C''\{\Xi''\}}\,(\text{Seq})}{\{\Xi \star \Theta\}C;\, C''\{\Xi'' \star \Theta\}}\,(\text{Fr }\Theta) \qquad \dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi'' \star \Theta\}C'\{\Xi'' \star \Theta'\}}\,(\text{Fr }\Xi'')}{\{\Xi \star \Theta\}C;\, C'';\, C'\{\Xi'' \star \Theta'\}}\,(\text{Seq})$$

Figure 9.9: Rewrite rule to improve temporal locality

Theorem 18. *The rewrite rule TemporalLocality is sound.*

Proof. Observe that the leaves of TemporalLocality's right-hand side are identical to the leaves of TemporalLocality's left-hand side (which are valid by hypothesis). Further observe that each inference in TemporalLocality's right-hand side is a valid instance of the proof rules. □

9.4.3 Examples of Optimization

Figure 9.10 shows procedure `copy_and_dispose` that copies the content of field *val* of cell x to field *val* of cell c. As indicated by its subscript, lock l has $c \mapsto [val : \_]$ as its invariant, i.e., it guards access to the cell pointed to by c. Applying GenericOptimization optimizes `copy_and_dispose` in two ways: the critical region is shortened and cell x is disposed earlier.

Figure 9.11 shows how to combine TemporalLocality and Parallelize. Figure 9.11 shows method `mth` that traverses two lists. The method for list traversal is not shown,

```
requires x ↦ [val : _];
ensures  emp;
copy_and_dispose(x;){
 local v;
 lock(l_{c↦[val:_]});
  v := x→val;
  c→val := v;
  dispose(x);
 unlock(l_{c↦[val:_]});
}
```

→

```
requires x ↦ [val : _];
ensures  emp;
copy_and_dispose(x;){
 local v;
 v := x→val;
 dispose(x);
 lock(l_{c↦[val:_]});
  c→val := v;
 unlock(l_{c↦[val:_]});
}
```

Figure 9.10: Optimization of a critical region

it could perform any action (reversal, sorting etc.) as long as it takes as input a list and outputs a list (as indicated by `list_traverse`'s contract). The optimization is performed in three steps: (1) FactorizeFrames is applied, (2) TemporalLocality is applied to execute the two traversals of list x in sequence, and (3) Parallelize is applied to parallelize the traversals on x and the traversal on y. Because there are no constraints at all on the implementation of list traversal, we believe this kind of optimization can be applied often in practice.

```
requires list(x);
ensures  list(x);
list_traverse(x;){ ... }
```

```
requires list(x) ⋆ list(y);
requires list(x) ⋆ list(y);
mth(x, y;){
 list_traverse (x;);
 list_traverse (y;);
 list_traverse (x;);
}
```

→

```
requires list(x);
ensures  list(x);
list_traverse(x;){ ... }
```

```
requires list(x) ⋆ list(y);
requires list(x) ⋆ list(y);
mth(x, y;){
 (list_traverse (x;)
  list_traverse (x;)) ||
 list_traverse (y;);
}
```

Figure 9.11: Optimization that combines TemporalLocality and Parallelize

9.5 Implementation

The techniques described in this chapter have been implemented in a tool called *éterlou*. Éterlou consists of two distinct modules:

A proof tree generator which is an extended version of smallfoot [15]. The proof tree generator generates proof trees using Figure 9.2's rules. Our extension does not interfere with the algorithms already present in smallfoot: it only computes antiframes and frames at each atomic command (by using both smallfoot's built-in algorithms and dedicated algorithms).

Because the (Frame) rule is the central ingredient of our procedure, it is crucial that the implementation of Figure 9.2's rules computes the *largest frames* (formulas Ξ_f) possible.

For example, our implementation of the rule for field lookup (Figure 9.2's second rule) computes the smallest antiframe Π_a that suffices to show $F = E$, and thus the largest frames.

A proof tree rewriter which implements the various rewrite rules shown in this chapter. The proof tree rewriter is written in tom [5], an extension of Java that adds constructs for pattern matching. We make extensive use of tom's mapping facility to pattern match against user-defined Java objects. Another crucial feature of tom is the possibility to define rewriting strategies.

All the examples of this chapter have been generated with éterlou. We have tested éterlou against several example programs provided in smallfoot's distribution and pointer programs of our own. Our experiments revealed that to obtain the best optimizations possible, the rewrite rules must be applied in a given order and/or with specific strategies. For example, FactorizeFrames must be applied before Parallelize for the latter rewrite rule to fire. In addition, applying rewrite rules from top to bottom (i.e., rewriting at the root before trying to rewrite in subtrees) generally yields programs where parallelized commands are longer (compared to other strategies such as bottom to top).

9.6 How to Take Advantage of Separation Logic's Advances

In this section, we review advances of separation logic that have not been implemented in smallfoot and we describe how our technique would benefit from these advances.

9.6.1 Object-Orientation

In object-oriented separation logic (as described from Chapter 2 to Chapter 5), separation logic's $\star$ splits objects per field. With smallfoot's notations, this means that $p \mapsto [x : \_] \star p \mapsto [y : \_]$ would represent a point with two fields x and y (and omitted fields are *not* existentially quantified). Splitting on a per-field basis provides more fine-grained parallelism which allows to build such a proof:

$$\begin{array}{c} \{p \mapsto [x : \_] \star p \mapsto [y : \_]\} \\ p \rightarrow x := E \parallel p \rightarrow y := F \\ \{p \mapsto [x : E] \star p \mapsto [y : F]\} \end{array}$$

Using a per-field semantics for $\star$ in the proof tree generator would allow Parallelize to fire more often. For example, in `rotate_tree` (see Figure 9.7), $x \rightarrow l := x_2$`;` $x \rightarrow r := x_1$ would be parallelized to $x \rightarrow l := x_2 \parallel x \rightarrow r := x_1$.

In addition, we emphasize that lifting our technique to object-oriented programs is straightforward since our procedure's key mechanism is the (Frame) rule which is supported by object-oriented separation logic (see Chapter 3 and Parkinson's work [88])

9.6.2 Permission Accounting

Permission accounting (see Chapter 3 to Chapter 5) gives an alternative reading of the points-to predicate $\mapsto$ by adding an extra parameter (called a *permission* π) to it. In the flavor of separation logic with permissions, one can define a predicate `tree(`t`,`π`)` representing access π to a tree t such that the following property holds:

$$\texttt{tree}(t, \pi) \Leftrightarrow \texttt{tree}(t, \frac{\pi}{2}) \star \texttt{tree}(t, \frac{\pi}{2}) \qquad \text{(split)}$$

$$\frac{\{\texttt{tree}(t, \tfrac{1}{2})\}\texttt{contains}(t,n)\{\texttt{tree}(t, \tfrac{1}{2})\} \qquad \{\texttt{tree}(t, \tfrac{1}{2})\}\texttt{contains}(t,m)\{\texttt{tree}(t, \tfrac{1}{2})\}}{\{\texttt{tree}(t, \tfrac{1}{2}) \star \texttt{tree}(t, \tfrac{1}{2})\}\texttt{contains}(t,n) \parallel \texttt{contains}(t,m)\{\texttt{tree}(t, \tfrac{1}{2}) \star \texttt{tree}(t, \tfrac{1}{2})\}} \text{ (Parallel)}$$

Figure 9.12: Proof tree of a program readonly accessing a tree in parallel

$$\frac{\{\texttt{tree}(t,1)\}\texttt{contains}(t,n)\{\texttt{tree}(t,1)\} \qquad \{\texttt{tree}(t,1)\}\texttt{contains}(t,n)\{\texttt{tree}(t,1)\}}{\{\texttt{tree}(t,1)\}\texttt{contains}(t,n)\texttt{; contains}(t,m)\{\texttt{tree}(t,1)\}} \text{ (Seq)}$$

Figure 9.13: Proof tree of a parallelizable program

Now, consider a procedure `contains(`t`,`n`)` that looks up if parameter n is in the set of values contained in tree t:

```
requires tree(t,π); ensures tree(t,π);
contains(t,n){ ... }
```

Method `contains`'s contract is parameterized by permission π. This means that `contains`'s precondition `tree(`t`,`π`)` can be instantiated by any π. This technique allows to verify programs where different processes simultaneously read access a tree as the proof tree in Figure 9.12 shows.

Now consider the proof tree in Figure 9.13. It is simple and depicts the typical reasoning performed in permission accounting separation logic. The rewrite rule Parallelize shown in Section 9.3.1, however, does not fire on this proof tree. To obtain Figure 9.12's proof tree by rewriting Figure 9.13's proof tree, we need to apply the (split) equivalence from left to right to be able to apply FactorizeFrames and Parallelize. This requires observing that `tree(`$t, 1$`)` can be split and that `contains`'s contract is parameterized. In another words, Figure 9.13's proof tree does not show that contains is a read-only method (because the `tree` predicate has `1` as second argument). For the parallelization to work, one should (1) infer that predicate `tree`'s second argument can be changed to $1/2$ and (2) insert applications of (Frame) accordingly.

We have not found an elegant solution to this issue yet but we observe that permissions accounting would allow Parallelize to fire strictly more often.

9.6.3 Fork/Join Parallelism

In Chapter 4, we showed how to handle dynamic thread creation with fork/join in separation logic. Recall that fork and join work as follows: (1) calling t`.fork()` starts a new thread t that executes in parallel with the rest of the program and (2) calling t`.join()` stops the calling thread until thread t finishes: when t finishes the calling thread is resumed.

When a parent thread forks a new thread, a part of the parent's state is transferred to the new thread. In smallfoot's language, this would be formalized by the following rule:

$$\frac{\Xi \text{ is } t\text{'s precondition}}{\{\Xi\}t.\texttt{fork()}\{\texttt{true} \mid \texttt{emp}\}} \text{ (Fork)}$$

Dually, when a thread joins another thread, the former "takes back" a part of the latter's state. To formalize this behavior, we have shown in Chapter 4 how to add a new predicate `Join(`t, π`)` which asserts that the thread in which it appears can take back part

π of thread t's state. In addition, the assertion language allows to multiply formulas by a permission, written $\pi \cdot \Xi$ (for simplicity, we do not reintroduce Chapter 4's predicates here). To give the reader an intuition of the meaning of multiplication, we note that integrating multiplication in smallfoot's framework would make the following property true:

$$\pi \cdot (\Pi \mid x \overset{1}{\mapsto} [\rho]) \Leftrightarrow \Pi \mid x \overset{\pi}{\mapsto} [\rho]$$

With the `Join` predicate and formula multiplication, one can formalize `join`'s behavior. For smallfoot, the rule below would express that a thread joining another thread t can take back a part of t's state:

$$\frac{\Xi \text{ is } t\text{'s postcondition}}{\{\texttt{Join}(t, \pi)\} t.\texttt{join}()\{\pi \cdot \Xi\}} \text{ (Join)}$$

Integrating (Fork) and (Join) in our framework would add two concrete applications to the rewrite rule GenericOptimization. (1) If C' is a `fork` command, GenericOptimization would rewrite proofs so that new threads are forked as soon as possible (increasing parallelism). (2) If C is a `join` command, GenericOptimization would rewrite proofs so that threads join other processes as late as possible (increasing parallelism and reducing joining time).

9.6.4 Variable as Resources

Bornat et al. [20] showed how to treat variables like heap cells. This allows to get rid of the side condition in the (Parallel) rule resulting in a more uniform proof system. To do this, the assertion language contains a new predicate $\texttt{Own}_\pi(x)$ that asserts ownership π of variable x.

Roughly, writing a variable x requires permission $\texttt{Own}_1(x)$ while reading a variable requires $\texttt{Own}_\pi(x)$ for any permission π (in analogy with the permission-accounting model). This is useful to rule out races on shared variables (whereas the permission system shown in Chapter 3 to Chapter 5 rules out races on heap cells) as the following program exemplifies:

$$\begin{array}{l} \{\texttt{Own}_1(x) \star ?\} \\ x = y \parallel x = z \end{array}$$

Above, ? cannot be filled with a predicate asserting ownership of x (needed for verifying the parallel statement's right-hand side) because $\texttt{Own}_1(x)$ is already needed to verify the parallel statement's left-hand side (and $\texttt{Own}_1(x)$ cannot be $\star$-combined with $\texttt{Own}_\pi(x)$ for any π).

The variable as resources technique would fit perfectly in our framework because variables that are not accessed by commands would be made explicit: $\texttt{Own}_\pi(\_)$ predicates would appear in frames. In other words, program verifiers implementing the variable as resources technique would compute explicit frames and antiframes for atomic commands like Figure 9.2's rules do. Finally, the variable as resource technique would allow for more uniform treatment of the Parallelize rewrite rule, because (Parallel)'s side condition would be deleted.

9.7 Future Work

Extending this chapter's procedure to object-oriented programs is the next step to prove this procedure's usefulness. Detractors for this procedure claim it is impractical for real world programs. I disagree: as long as there is a $\star$ operator in the specification language and applications of the (Frame) rule in proof trees, this procedure is surprisingly effective (as the implementation witnesses). To craft this extension, one would need to integrate this procedure in a verifier for object-oriented programs with separation logic [43, 33]. Another possible future work would be to apply this technique to parallelize loops. This would permit comparisons with traditional parallelizers [19, 4, 2, 100] that are effective at parallelizing loops.

Finally, because FactorizeFrames's guard (see page 161) uses the syntactical equivalence $\Leftrightarrow$, it might miss some semantical equivalences. Using an entailment relation $\vdash$ would be more powerful. However, we leave the problem of finding common frames with a semantical equivalence as future work for the following reason: finding a common frame (i.e., given Ξ and Θ; find Ξ_c, Ξ_r, and Θ_r such that $\Xi \vdash \Xi_r \star \Xi_c$ and $\Theta \vdash \Theta_r \star \Xi_c$) cannot be expressed efficiently in terms of known problems; such as a frame problem [16] (given Ξ and Θ, find Ξ_f such that $\Xi \vdash \Xi_f \star \Theta$), or a bi-abduction problem [29] (given Ξ and Θ, find Ξ_a and Θ_f such that $\Xi \star \Xi_a \vdash \Theta \star \Theta_f$).

9.8 Related Work and Conclusion

Related Work. The closest (and concurrent) related work is by Raza et al. [92] They use separation logic to parallelize programs. Our work differs in four ways: (1) Raza et al. attaches labels to heaps and uses disjointness of labels to detect possible parallelism, while we use the (Frame) rule to statically detect possible parallelism, leading to a technically simpler procedure; (2) we express optimizations by rewrite rules on proof trees, allowing us to feature other optimizations than parallelization and to use different optimization strategies; (3) Raza et al. is applied after a shape analysis [42, 13], while our analysis is applied after verification with a program verifier; and (4) contrary to Raza et al., we have an implementation.

Practical approaches for parallelizing programs include parallelizing compilers [19, 4]. Parallelizing compilers focuses on loop parallelization and do not consider arbitrary pieces of code. Parallelizing compilers can yield code that executes an order of magnitude faster than classical compilers. Loop parallelization has been actively studied [75, 2, 100].

Formal approaches for optimizing programs include certified compilers [97, 78] and certifying compilers [11, 82]. Certified compilers include optimizations that we do not consider and provide fully machine-checked proofs. Certifying compilers manipulate formulas representing proof obligations whereas we manipulate proof trees representing derivation of Hoare triples. For this reason, we can consider high-level optimizations such as parallelization whereas we cannot consider the low-level optimizations described in [11, 82].

Techniques to dispose memory as soon as possible have been studied for machine registers [46] where the goal is to use as few registers as possible. Works on atomicity [38, 22] include techniques to release locks as soon as possible. Improving temporal locality has been studied for a particular type of programs [71].

Conclusion We showed a new technique to optimize programs proven correct in separation logic. Proofs are represented as a derivation of Hoare triples. The core of the

procedure uses separation logic's (Frame) rule to statically detect parts of the state which are useless for a command to execute. Considered optimizations are parallelization, early disposal, late allocation, early lock releasing, and late lock acquirement. Optimizations are expressed as rewrite rules between proof trees and are performed automatically.

The procedure has been implemented in the éterlou tool. Éterlou consists of a proof tree generator (a modified version of the smallfoot program verifier [15]) and a proof tree rewriter written in tom [5]. Small-scale experiments show that the approach is practical.

Chapter 10

Conclusion

> Je ne suis pas seulement non violent mais je suis pour la non-puissance. Ce n'est sûrement pas une technique efficace. [...] [Mais] on ne peut pas créer une société juste avec des moyens injustes. On ne peut pas créer une société libre avec des moyens d'esclaves. C'est pour moi le centre de ma pensée.
>
> *Entretiens (1994)*
> Jacques Ellul

This thesis presented several extensions to separation logic for multithreaded object-oriented programs. This work builds on separation logic [93] for while programs, fractional permissions [23, 21], concurrent separation logic [84], and separation logic for object-oriented programs [88]. Further, this thesis showed three new analyzes based on separation logic. Below, we list the contributions of this thesis and discuss possible future work.

Contributions. From Chapter 2 to Chapter 7, we showed a verification system that handles fundamental aspects of multithreaded Java-like programs.

This verification system handles dynamic thread creation, i.e., fork/join style of parallelism (Chapter 4) à la Java. Further, this verification system supports Java's reentrant locks (Chapter 5). We provide several new features to extend expressiveness: parameterized classes and methods, and support for multiple readonly joiner threads. The applicability of this verification system has been checked on design patterns from the literature. Examples include the worker thread pattern and a lock coupling algorithm. Importantly, this verification system has been shown sound (Chapter 6).

In Chapters 7, 8, and 9, we described three new analyzes based on separation logic.

We showed how to specify protocols of multithreaded Java-like programs (Chapter 7). This work is an extension of Cheon et al.'s work [32]. In addition, we described a new technique to show that method contracts are correct w.r.t. protocols. For this, we generate programs that must be proven correct. If the generated programs cannot be proven correct, some client programs will fail to verify even if they obey the underlying protocol. Because this technique can be applied early in the development process and helps writing correct specifications (which are harder to test than programs), we believe it is a valuable tool for developers using formal methods.

We presented a new technique to disprove entailment between separation logic formulas (Chapter 8). We abstract models of formulas by their size and check whether two formulas have models whose sizes are compatible. Given two formulas A and B that do not have compatible models, we can conclude that $A \not\models B$. Because of its low complexity, our algorithm is of interest wherever (1) entailment checking is performed often or (2) entailment checking is undecidable (as in Chapters 3 to 5's variant of separation logic).

Finally, we showed a new technique to automatically parallelize and optimize programs by rewriting their proofs (Chapter 9). The core of the procedure uses separation logic's (Frame) rule to statically detect parts of the state which are useless for a command to execute. Transformations are expressed as rewrite rules between proof trees and are performed automatically. The procedure has been implemented in the éterlou tool.

Future Work. First, it is unclear if the contracts of `run`, `fork`, and `join` shown in Chapter 4 are the most convenient ones. Because `run`'s contract should be expressed in the definitions of predicates `preFork` and `postJoin`, it might be inconvenient to specify `run` when several `Thread` classes inherit from each other. This issue did not occur in examples of this thesis, but we believe a broader study would be valuable. As sketched in Section 4.4, a more flexible solution would be to add scalar multiplication as a new constructor for formulas (as studied by Boyland [24]).

Second, the fact that proving the precondition of the rule for acquiring reentrant locks (see rule (Re-Lock) in Chapter 5) requires solving an aliasing problem is unsatisfying. While we believe this is unavoidable with specifications that do not mention thread identifiers, it might be that specifications that include thread identifiers would avoid this problem (at the cost of putting an extra burden on the programmer).

Third, in Chapters 4 and 5, we do not verify deadlock freeness. Because deadlocks depend on the scheduler, they are hard to reproduce and to eliminate manually. Consequently, we believe it is important to provide formal support to avoid them. Deadlock freeness has been studied extensively in the literature, in particular within general verification systems [35, 69, 77]. We emphasize that, to be complete (as opposed to the works just cited), also deadlocks caused by join must be avoided. Join-based deadlocks occur when two threads join each other. Traditional solutions to deadlocks (such as imposing an ordering relation on the locks acquired) should be adapted to fix this issue (e.g., by ordering the threads joined).

Fourth, it would be valuable to implement the verification system described from Chapter 3 to 5. In particular, it is unclear if this system can be completely automated. There are three problems that should be tackled to reach this goal: (1) when to open and close abstract predicates ? (2) how to automate entailment checking (in presence of the magic wand it becomes undecidable) ? and (3) how to combine type-based reasoning and separation logic reasoning (as exemplified in Section 5.6.3's lock coupling example) ?

Fifth, Chapter 8's procedure to disprove entailment and Chapter 9's procedure for automatic parallelization could be adapted to an object-oriented language. While this requires developing new algorithms for Chapter 8's procedure, it should be straightforward for Chapter 9's procedure.

Appendix A

Auxiliary Definitions for Chapters 2 to 7

Field Lookup, $\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) = \bar{T}\ \bar{f}$**:**

$$\frac{}{\mathsf{fld}(\texttt{Object}) = \emptyset}\ (\text{Fields Base}) \qquad \frac{\mathsf{fld}(D\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}) = \bar{T}'\ \bar{f}' \quad \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ D\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{impl}\ \bar{U}\ \texttt{\{}\bar{T}\ \bar{f}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}}{\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) = (\bar{T}\ \bar{f})[\bar{\pi}/\bar{\alpha}],\ \bar{T}'\ \bar{f}'}\ (\text{Fields Ind})$$

Axiom Lookup, $\mathsf{axiom}(t\texttt{<}\bar{\pi}\texttt{>}) = F$**:**

$$\mathsf{axiom}(ax\texttt{*}) \triangleq \begin{cases} \texttt{true} & \text{if } ax\texttt{*} = () \\ F * \mathsf{axiom}(ax\texttt{*}) & \text{if } ax\texttt{*} = (\texttt{axiom}\ F, ax\texttt{*}) \end{cases}$$

$$\mathsf{axiom}(\bar{T}) \triangleq \begin{cases} \texttt{true} & \text{if } \bar{T} = () \text{ or } \bar{T} = (\texttt{Object}) \\ \mathsf{axiom}(U) * \mathsf{axiom}(\bar{V}) & \text{if } \bar{T} = (U, \bar{V}) \end{cases}$$

$$\frac{\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}}{\mathsf{axiom}(C\texttt{<}\bar{\pi}\texttt{>}) = \mathsf{axiom}(ax\texttt{*}[\bar{\pi}/\bar{\alpha}]) * \mathsf{axiom}((U, \bar{V})[\bar{\pi}/\bar{\alpha}])}\ (\text{Ax Class})$$

$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}}{\mathsf{axiom}(I\texttt{<}\bar{\pi}\texttt{>}) = \mathsf{axiom}(ax\texttt{*}[\bar{\pi}/\bar{\alpha}]) * \mathsf{axiom}(\bar{U}[\bar{\pi}/\bar{\alpha}])}\ (\text{Ax Interface})$$

Remarks on method lookup (defined below):

- In mbody and mtype, we replace the implicit self-parameter `this` by an explicit method parameter (separated from the other method parameters by a semicolon). This is technically convenient for the theory.
- In mtype, we replace the implicit result-parameter `result` by an explicit existential quantifier over the postcondition. This is technically convenient for the theory.

Method Lookup, $\mathsf{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = mt$ **and** $\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\bar{\imath}).c$**:**

(Mlkup `Object`)
$$\frac{\texttt{class Object \{}\ldots\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}\ \ldots\texttt{\}}}{\mathsf{mlkup}(m, \texttt{Object}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}}$$

(Mlkup Defn)
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U'\ \texttt{impl}\ \bar{V}'\ \texttt{\{}\ldots\ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}\ \ldots\texttt{\}}}{\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}})[\bar{\pi}/\bar{\alpha}']}$$

(Mlkup Inherit) $\quad m \notin \mathsf{dom}(md\texttt{*})$
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ D\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{impl}\ \bar{U}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*\}} \quad \mathsf{mlkup}(m, D\texttt{<}\bar{\pi}'[\bar{\pi}/\bar{\alpha}]\texttt{>}) = md'}{\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = md'}$$

If $\mathsf{mlkup}(m, C\texttt{<}\bar{\pi}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)\{}c\texttt{\}}$, then:
$$\mathsf{mbody}(m, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq (\texttt{this};\bar{\imath}).c$$
$$\mathsf{mtype}(m, C\texttt{<}\bar{\pi}\texttt{>}) \triangleq \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\ F\texttt{;}\ \texttt{ensures (ex}\ U\ \texttt{result)(}G\texttt{);}\ U\ m\texttt{(}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{this;}\ \bar{V}\ \bar{\imath}\texttt{)}$$

(Mtype Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\ldots\ \texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{requires}\ F\texttt{;}\ \texttt{ensures}\ G\texttt{;}\ U'\ m\texttt{(}\bar{V}'\ \bar{\imath}\texttt{);}\ \ldots\texttt{\}}}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{requires}\ F\texttt{;}\ \texttt{ensures (ex}\ U'\ \texttt{result)(}G\texttt{);}\ U'\ m\texttt{(}I\texttt{<}\bar{\pi}\texttt{>}\ \texttt{this;}\ \bar{V}'\ \bar{\imath}\texttt{))}[\bar{\pi}/\bar{\alpha}]}$$

(Mtype Interface Inherit) $\quad \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}, V, \bar{U}'\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$
$$\frac{m \notin \mathsf{dom}(mt\texttt{*}) \quad (\forall U \in \bar{U}, \bar{U}')(\mathsf{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{mtype}(m, V[\bar{\pi}/\bar{\alpha}]) = mt}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = mt}$$

(Mtype Interface Inherit `Object`) $\quad \texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$
$$\frac{m \notin \mathsf{dom}(mt\texttt{*}) \quad (\forall U \in \bar{U})(\mathsf{mtype}(m, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{mtype}(m, \texttt{Object}) = mt}{\mathsf{mtype}(m, I\texttt{<}\bar{\pi}\texttt{>}) = mt}$$

Remarks on predicate lookup:

- The "ext `Object`" in $\mathsf{plkup}(\texttt{init}, \texttt{Object})$ and (Plkup `Object`) is included to match the format of the relation. There is nothing more to this.
- Each class implicitly defines the `init`-predicate, which gives write permission to all fields of the class frame. In (Plkup `init`), $\mathsf{df}(T)$ is the default value of type T (df is formally defined in Section 2.2).

Predicate Lookup, $\mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>}) = pt$ **and** $\mathsf{pbody}(\pi.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F\ \mathsf{ext}\ T$**:**

$$\mathsf{plkup}(\texttt{init}, \texttt{Object}) = \texttt{pred init = true ext Object}$$

(Plkup `Object`)
$$\frac{\texttt{class Object \{}\ldots\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\texttt{;}\ \ldots\texttt{\}}}{\mathsf{plkup}(P, \texttt{Object}) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\ \mathsf{ext}\ \texttt{Object}}$$

(Plkup Defn)
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\ldots\ \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\texttt{;}\ \ldots\texttt{\}}}{\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\ \mathsf{ext}\ \texttt{Object})[\bar{\pi}/\bar{\alpha}']}$$

(Plkup `init`)
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*\}} \quad F = \circledast_{T\ f \in fd}\texttt{*PointsTo(this.}f\texttt{,}1\texttt{,}\mathsf{df}(T)\texttt{)}}{\mathsf{plkup}(\texttt{init}, C\texttt{<}\bar{\pi}\texttt{>}) = (\ \texttt{pred init = }F\ \texttt{ext}\ U\ )[\bar{\pi}/\bar{\alpha}']}$$

(Plkup Inherit) $P \notin \mathsf{dom}(pd\texttt{*})$
$$\frac{\texttt{class}\ C\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ md\texttt{*\}} \quad \mathsf{plkup}(P, U) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\ \texttt{ext}\ U'}{\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = \texttt{true}\ \texttt{ext}\ U)[\bar{\pi}/\bar{\alpha}']}$$

If $\mathsf{plkup}(P, C\texttt{<}\bar{\pi}\texttt{>}) = \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} = F\texttt{ext}\ V$, then:
$$\begin{aligned} \mathsf{pbody}(\pi.P\texttt{<}\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}\texttt{>}) &\triangleq (F\ \texttt{ext}\ V)[\pi/\texttt{this}, \bar{\pi}'/\bar{\alpha}] \\ \mathsf{ptype}(P, C\texttt{<}\bar{\pi}\texttt{>}) &\triangleq \texttt{pred}\ P\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>} \end{aligned}$$

(Ptype Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\ldots\ \texttt{pred}\ P\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>;}\ \ldots\texttt{\}}}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = (\texttt{pred}\ P\texttt{<}\bar{T}'\ \bar{\alpha}'\texttt{>})[\bar{\pi}/\bar{\alpha}]}$$

(Ptype Interface Inherit) $\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}, V, \bar{U}'\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$
$$\frac{P \notin \mathsf{dom}(\mathcal{S}) \quad (\forall U \in \bar{U}, \bar{U}')(\mathsf{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{ptype}(P, V[\bar{\pi}/\bar{\alpha}]) = pt}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = pt}$$

(Ptype Interface Inherit `Object`) $\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$
$$\frac{P \notin \mathsf{dom}(pt\texttt{*}) \quad (\forall U \in \bar{U})(\mathsf{ptype}(P, U[\bar{\pi}/\bar{\alpha}]) = \mathsf{undef}) \quad \mathsf{ptype}(P, \texttt{Object}) = pt}{\mathsf{ptype}(P, I\texttt{<}\bar{\pi}\texttt{>}) = pt}$$

The partial function $\mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>})$ is extended to predicate selectors $P\texttt{@}C$ as follows:
$$\mathsf{ptype}(P\texttt{@}C, t\texttt{<}\bar{\pi}\texttt{>}) \triangleq \begin{cases} \mathsf{ptype}(P, t\texttt{<}\bar{\pi}\texttt{>}) & \text{if } t = C \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Protocol Lookup, $\mathsf{protlkup}(C\texttt{<}\bar{\pi}\texttt{>}) = \bar{s}$

(Protlkup `Object`)
$$\frac{}{\mathsf{protlkup}(\texttt{Object}) = ()}$$

(Protlkup Class)
$$\frac{\mathsf{protlkup}(U) = s' \quad \mathsf{protlkup}(\bar{V}) = s'' \quad \texttt{class}\ C\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \bar{V}\ \texttt{\{}\ldots\ \texttt{protocol}\ s\texttt{;}\ \ldots\texttt{\}}}{\mathsf{protlkup}(C\texttt{<}\bar{\pi}\texttt{>}) = (s, s', s'')[\bar{\pi}/\bar{\alpha}]}$$

(Protlkup Interface) $\mathsf{protlkup}(\bar{U}) = s'$
$$\frac{\texttt{interface}\ I\texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{ext}\ \bar{U}\ \texttt{\{}\ldots\ \texttt{protocol}\ s\texttt{;}\ \ldots\texttt{\}}}{\mathsf{protlkup}(I\texttt{<}\bar{\pi}\texttt{>}) = (s, s')[\bar{\pi}/\bar{\alpha}]}$$

Appendix B

Hoare Triples for Java-like Programs

$$\frac{C\texttt{<}\bar{T}\,\bar{\alpha}\texttt{>} \in ct \quad \Gamma \vdash \bar{\pi} : \bar{T}[\bar{\pi}/\alpha] \quad C\texttt{<}\bar{\pi}\texttt{>} <: \Gamma(\ell)}{\Gamma; v \vdash \begin{array}{c}\{\texttt{true}\} \\ \ell \texttt{ = new } C\texttt{<}\bar{\pi}\texttt{>} \\ \{\ell\texttt{.init} * C \texttt{ classof } \ell * \circledast_{\Gamma(u)<:\texttt{Object}}\, \ell \texttt{ != } u * \ell\texttt{.fresh}\}\end{array}} \text{ (New)}$$

$$\frac{\Gamma \vdash \pi, \pi' : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\texttt{Lockset}(\pi') * \pi\texttt{.inv} * \pi\texttt{.fresh}\} \\ \pi\texttt{.commit} \\ \{\texttt{Lockset}(\pi') * \texttt{!}(\pi' \texttt{ contains } \pi) * \pi\texttt{.initialized}\}\end{array}} \text{ (Commit)}$$

Figure B.1: Rules for initialization commands

$$\frac{\Gamma \vdash u, w : U, W \quad W\ f \in \mathsf{fld}(U)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f, 1, W)\}u.f = w\{\texttt{PointsTo}(u.f, 1, w)\}} \text{ (Fld Set)}$$

$$\frac{\Gamma \vdash u, \pi, w : U, \texttt{perm}, W \quad W\ f \in \mathsf{fld}(U) \quad W <: \Gamma(\ell)}{\Gamma; v \vdash \{\texttt{PointsTo}(u.f, \pi, w)\}\ell = u.f\{\texttt{PointsTo}(u.f, \pi, w) * \ell \texttt{ == } w\}} \text{ (Get)}$$

Figure B.2: Rules for commands that access the heap

$$\frac{\begin{array}{c}\mathsf{mtype}(m, t\texttt{<}\bar{\pi}\texttt{>}) = \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>}\ \texttt{requires}\, G; \texttt{ensures (ex}\, U\ \alpha')\texttt{(}G'\texttt{)};\ U\ m\texttt{(}t\texttt{<}\bar{\pi}\texttt{>}\, \imath_0; \bar{W}\ \bar{\imath}\texttt{)} \\ \sigma = (u/\imath_0, \bar{\pi}'/\bar{\alpha}, \bar{w}/\bar{\imath}) \quad \Gamma \vdash u, \bar{\pi}', \bar{w} : t\texttt{<}\bar{\pi}\texttt{>}, \bar{T}[\sigma], \bar{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma; v \vdash \{u\ \texttt{!= null *}\ G[\sigma]\}\ell = u.m(\bar{w})\{\texttt{(ex}\, U[\sigma]\ \ \alpha')\texttt{(}\alpha'\ \texttt{==}\ \ell\ \texttt{*}\ G'[\sigma]\texttt{)}\}}\ \text{(Call)}$$

$$\frac{\Gamma \vdash u, S : \texttt{Thread}, \texttt{lockset}}{\Gamma; v \vdash \{u\ \texttt{!= null *}\ \texttt{Lockset(}S\texttt{)}\ \texttt{*}\ u.\texttt{preFork}\}\ell = \texttt{u.fork()}\{\texttt{Lockset(}S\texttt{)}\}}\ \text{(Call instantiated by } \texttt{fork}\text{)}$$

$$\frac{\Gamma \vdash u, p : \texttt{Thread}, \texttt{perm}}{\Gamma; v \vdash \{u\ \texttt{!= null *}\ \texttt{Join}(u, p)\}\ell = u.\texttt{join()}\{u.\texttt{postJoin<p>}\}}\ \text{(Call instantiated by } \texttt{join}\text{)}$$

Figure B.3: Rules for method calls

$$\frac{\Gamma \vdash u, \pi : \texttt{Object}, \texttt{lockset}}{\begin{array}{c}\Gamma; v \vdash \{\texttt{Lockset(}\pi\texttt{)}\ \texttt{*}\ \texttt{!(}\pi\ \texttt{contains}\ u\texttt{)}\ \texttt{*}\ u.\texttt{initialized}\} \\ u.\texttt{lock()} \\ \{\texttt{Lockset(}u \cdot \pi\texttt{)}\ \texttt{*}\ u.\texttt{inv}\}\end{array}}\ \text{(Lock)}$$

$$\frac{\Gamma \vdash u.\pi : \texttt{Object}, \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset(}u \cdot \pi\texttt{)}\}u.\texttt{lock()}\{\texttt{Lockset(}u \cdot u \cdot \pi\texttt{)}\}}\ \text{(Re-Lock)}$$

$$\frac{\Gamma \vdash u : \texttt{Object} \quad \Gamma \vdash \pi : \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset(}u \cdot u \cdot \pi\texttt{)}\}u.\texttt{unlock()}\{\texttt{Lockset(}u \cdot \pi\texttt{)}\}}\ \text{(Re-Unlock)}$$

$$\frac{\Gamma \vdash u : \texttt{Object} \quad \Gamma \vdash \pi : \texttt{lockset}}{\Gamma; v \vdash \{\texttt{Lockset(}u \cdot \pi\texttt{)}\ \texttt{*}\ u.\texttt{inv}\}u.\texttt{unlock()}\{\texttt{Lockset(}\pi\texttt{)}\}}\ \text{(Unlock)}$$

Figure B.4: Rules for locking/unlocking

$$\frac{\Gamma; v; F \vdash G[w/\alpha] \quad \Gamma \vdash w : U <: T \quad \Gamma, \alpha : U \vdash G : \diamond}{\Gamma; v \vdash \{F\} w : T\{(\mathtt{ex}\ U\ \alpha)\,(G)\}} \text{ (Val)}$$

$$\frac{\ell \notin F, G \quad \Gamma, \ell : T; v \vdash \{F * \ell == \mathsf{df}(T)\} c : U\{G\}}{\Gamma; v \vdash \{F\} T\ \ell;\ c : U\{G\}} \text{ (Dcl)}$$

$$\frac{\begin{array}{l}\imath \notin F, G, v \\ \Gamma \vdash \ell : T \quad \Gamma, \imath : T; v \vdash \{F * \imath == \ell\} c : U\{G\}\end{array}}{\Gamma; v \vdash \{F\} T\ \imath = \ell;\ c : U\{G\}} \text{ (Fin Dcl)}$$

$$\frac{\Gamma; v \vdash \{F\} hc \{F'\} \quad \Gamma; v \vdash \{F'\} c : T\{G\}}{\Gamma; v \vdash \{F\} hc; c : T\{G\}} \text{ (Seq)}$$

$$\frac{\Gamma; v \vdash \{F\} hc \{G\} \quad \Gamma \vdash H : \diamond \quad \mathsf{fv}(H) \cap \mathsf{writes}(hc) = \emptyset}{\Gamma; v \vdash \{F * H\} hc \{G * H\}} \text{ (Frame)}$$

$$\frac{\begin{array}{l}\Gamma; v \vdash \{F'\} hc \{G'\} \\ \Gamma; v; F \vdash F' \quad \Gamma; v; G' \vdash G\end{array}}{\Gamma; v \vdash \{F\} hc \{G\}} \text{ (Consequence)} \qquad \frac{\Gamma, \alpha : T; v \vdash \{F\} hc \{G\}}{\Gamma; v \vdash \{(\mathtt{ex}\ T\ \alpha)\,(F)\} hc \{(\mathtt{ex}\ T\ \alpha)\,(G)\}} \text{ (Exists)}$$

$$\frac{\Gamma \vdash w : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = w \{\ell == w\}} \text{ (Var Set)} \qquad \frac{\Gamma \vdash op(\bar{w}) : \Gamma(\ell)}{\Gamma; v \vdash \{\mathtt{true}\} \ell = op(\bar{w}) \{\ell == op(\bar{w})\}} \text{ (Op)}$$

$$\frac{\begin{array}{l}\Gamma \vdash w : \mathtt{bool} \\ \Gamma; v \vdash \{F * w\} c : \mathtt{void}\{G\} \quad \Gamma; v \vdash \{F * \mathtt{!}w\} c' : \mathtt{void}\{G\}\end{array}}{\Gamma; v \vdash \{F\} \mathtt{if}\,(w)\{c\}\mathtt{else}\{c'\}\{G\}} \text{ (If)}$$

$$\frac{\Gamma; v; F \vdash G}{\Gamma; v \vdash \{F\} \mathtt{assert}(G)\{F\}} \text{ (Assert)}$$

$$\frac{\Gamma \vdash v : T \quad \Gamma; o; F \vdash G[v/\alpha] \quad T <: U \quad \Gamma, \ell : U; p \vdash \{(\mathtt{ex}\ T\ \alpha)\,(\alpha == \ell\ *\ G)\} c : V\{H\}}{\Gamma, \ell : U; o \vdash \{F\} \ell = \mathtt{return}(v);\ c : V\{H\}} \text{ (Return)}$$

Figure B.5: Other rules

Appendix C

Implementation of Chapter 9's Rewrite Rules

For the rewrite rules to work in practice, the implementation of the rewrite rules is more complex than the rules shown in Chapter 9. The caveat is the following: (1) the implementation expects that there is a continuation unaffected by the optimizations, (2) the implementation preserves the invariant that the first premise of an application of (Seq) is always an application of (Frame) and, (3) the implementation preserves the invariant that there are no double (Frame)s i.e., two applications of (Frame) on top of each other.

Rule (1) ensures that, given C and C' that can be parallelized, the Parallelize rule will fire on the program $C; C'; C''$ (and similarly for other optimizations). For rule (1) to allow optimizations to fire when there is *no* continuation, the rewrite rules insert dummy `empty` continuation in appropriate places. Rules (2) and (3) ensure that rewrite rules can assume proof trees have a given shape (without the need for investigating different cases). Rule (2)'s invariant that the first premise of an application of (Seq) is always a (Frame) is ensured by inserting "dummy" (Frame)s $\texttt{true} \upharpoonright \texttt{emp}$ in appropriate places. Rule (3)'s invariant that there are no double (Frame)s is ensured by $\star$-merging double (Frame)s when needed.

In the implementation of the rewrite rules, for space reasons, we write ϵ for `empty`. FactorizeFrames's implementation is shown in Figure C.1 and Parallelize in Figure C.2. GenericOptimization's implementation is exactly as shown in Section 9.4.1 (see Figure 9.8 on page 164).

The implementation features two variations of TemporalLocality. Figure C.3 shows a version meant to be applied before FactorizeFrames while Figure C.4 shows a version meant to be applied after FactorizeFrames. That is because FactorizeFrames takes a proof tree balanced to the right and yields a proof tree balanced to the left. To be applied before FactorizeFrames, the first version of TemporalLocality (Figure C.3) takes as input a proof tree balanced to the right, while the second version of TemporalLocality (Figure C.4) takes as input a proof tree balanced to the left. In practice, both versions of TemporalLocality are useful.

$$\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_f\}C\{\Xi_p \star \Xi_f\}}\,(\text{Frame } \Xi_f) \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_f\}C'\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Theta_f) \qquad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Theta_a \star \Theta_f\}C';\ C''\{\Xi'\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_f\}C;\ C';\ C''\{\Xi'\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{FactorizeFrames}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi_a\}C\{\Xi_p\}}{\{\Xi_a \star \Xi_{f_0}\}C\{\Xi_p \star \Xi_{f_0}\}}\,(\text{Frame } \Xi_{f_0}) \qquad \dfrac{\dfrac{\{\Theta_a\}C'\{\Theta_p\}}{\{\Theta_a \star \Theta_{f_0}\}C'\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Frame } \Theta_{f_0}) \qquad \dfrac{\Theta_p \star \Theta_{f_0} \vdash \Theta_p \star \Theta_{f_0}}{\{\Theta_p \star \Theta_{f_0}\}\epsilon\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Empty})}{\{\Theta_a \star \Theta_{f_0}\}C';\ \epsilon\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Seq})}{\{\Xi_a \star \Xi_{f_0}\}C;\ C';\ \epsilon\{\Theta_p \star \Theta_{f_0}\}}\,(\text{Seq}) \qquad \dfrac{\dfrac{\{\Xi_a \star \Xi_{f_0}\}C;\ C';\ \epsilon\{\Theta_p \star \Theta_{f_0}\}}{\{\Xi_a \star \Xi_f\}C;\ C';\ \epsilon\{\Theta_p \star \Theta_f\}}\,(\text{Frame } \Xi_c) \qquad \{\Theta_p \star \Theta_f\}C''\{\Xi'\}}{\{\Xi_a \star \Xi_f\}C;\ C';\ \epsilon;\ C''\{\Xi'\}}\,(\text{Seq})}\,(\text{Seq})$$

Guard: $\Xi_f \Leftrightarrow \Xi_{f_0} \star \Xi_c$ and $\Theta_f \Leftrightarrow \Theta_{f_0} \star \Xi_c$

Figure C.1: FactorizeFrames's implementation

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\}}{\{\Xi \star \Xi'\}C\{\Theta \star \Xi'\}}\,(\text{Frame } \Xi') \qquad \dfrac{\dfrac{\{\Xi'\}C'\{\Theta'\}}{\{\Theta \star \Xi'\}C'\{\Theta \star \Theta'\}}\,(\text{Frame } \Theta) \qquad \{\Theta \star \Theta'\}C''\{\Xi''\}}{\{\Theta \star \Xi'\}C';\ C''\{\Xi''\}}\,(\text{Seq})}{\{\Xi \star \Xi'\}C;\ C';\ C''\{\Xi''\}}\,(\text{Seq})$$

$$\downarrow \underline{(\text{Parallelize})}$$

$$\dfrac{\dfrac{\{\Xi\}C\{\Theta\} \qquad \{\Xi'\}C'\{\Theta'\}}{\{\Xi \star \Xi'\}C \| C'\{\Theta \star \Theta'\}}\,(\text{Parallel}) \qquad \{\Theta \star \Theta'\}C''\{\Xi''\}}{\{\Xi \star \Xi'\}(C \| C');\ C''\{\Xi''\}}\,(\text{Seq})$$

Figure C.2: Parallelize's implementation

$$\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}}{\{\Xi\star\Theta\}C\{\Xi'\star\Theta\}}\,(\text{Fr }\Theta)\qquad \dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi'\star\Theta\}C'\{\Xi'\star\Theta'\}}\,(\text{Fr }\Xi')\qquad \dfrac{\dfrac{\{\Xi'\}C''\{\Xi''\}}{\{\Xi'\star\Theta'\}C'';\ C'''\{\Xi''\star\Theta'\}}\,(\text{Fr }\Theta')\qquad \{\Xi''\star\Theta'\}C'''\{\Xi_p\}}{\{\Xi'\star\Theta'\}C'';\ C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi'\star\Theta\}C';\ C'';\ C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\ C';\ C'';\ C'''\{\Xi_p\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{TemporalLocality}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}\qquad \dfrac{\{\Xi'\}C''\{\Xi''\}\qquad \dfrac{\Xi''\vdash\Xi''}{\{\Xi''\}\epsilon\{\Xi''\}}\,(\text{Empty})}{\{\Xi'\}C'';\ \epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\}C;\ C'';\ \epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\ C'';\ \epsilon\{\Xi''\star\Theta\}}\,(\text{Fr }\Theta)\qquad \dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi''\star\Theta\}C'\{\Xi''\star\Theta'\}}\,(\text{Fr }\Xi'')\qquad \{\Xi''\star\Theta'\}C'''\{\Xi_p\}}{\{\Xi''\star\Theta\}C';\ C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\}C;\ C'';\ \epsilon;\ C';\ C'''\{\Xi_p\}}\,(\text{Seq})$$

Figure C.3: TemporalLocality's implementation to be applied before FactorizeFrames

$$\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}}{\{\Xi\star\Theta\star\Theta''\}C\{\Xi'\star\Theta\star\Theta''\}}\,(\text{Fr }\Theta\star\Theta'')\qquad \dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi'\star\Theta\}C'\{\Xi'\star\Theta'\}}\,(\text{Fr }\Xi')\qquad \dfrac{\dfrac{\{\Xi'\}C''\{\Xi''\}}{\{\Xi'\star\Theta'\}C''\{\Xi''\star\Theta'\}}\,(\text{Fr }\Theta')\qquad \dfrac{\Xi''\star\Theta'\vdash\Xi''\star\Theta'}{\{\Xi''\star\Theta'\}\epsilon\{\Xi''\star\Theta'\}}\,(\text{Empty})}{\{\Xi'\star\Theta'\}C'';\ \epsilon\{\Xi''\star\Theta'\}}\,(\text{Seq})}{\{\Xi'\star\Theta'\}C';\ C'';\ \epsilon\{\Xi''\star\Theta'\}}\,(\text{Seq})\qquad \dfrac{\{\Xi'\star\Theta\star\Theta''\}C';\ C'';\ \epsilon\{\Xi''\star\Theta'\star\Theta''\}}{\{\Xi'\star\Theta\star\Theta''\}C';\ C'';\ \epsilon;\ C'''\{\Xi_p\}}\,(\text{Fr }\Theta'')\qquad \{\Xi''\star\Theta\star\Theta''\}C'''\{\Xi_p\}}{\{\Xi\star\Theta\star\Theta''\}C;\ C';\ C'';\ \epsilon;\ C'''\{\Xi_p\}}\,(\text{Seq})$$

$$\downarrow \underline{\text{TemporalLocality}}$$

$$\dfrac{\dfrac{\dfrac{\{\Xi\}C\{\Xi'\}\qquad \dfrac{\{\Xi'\}C''\{\Xi''\}\qquad \dfrac{\Xi''\vdash\Xi''}{\{\Xi''\}\epsilon\{\Xi''\}}\,(\text{Empty})}{\{\Xi'\}C'';\ \epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\}C;\ C'';\ \epsilon\{\Xi''\}}\,(\text{Seq})}{\{\Xi\star\Theta\star\Theta''\}C;\ C'';\ \epsilon\{\Xi''\star\Theta\star\Theta''\}}\,(\text{Fr }\Theta\star\Theta'')\qquad \dfrac{\dfrac{\{\Theta\}C'\{\Theta'\}}{\{\Xi''\star\Theta\star\Theta''\}C'\{\Xi''\star\Theta'\star\Theta''\}}\,(\text{Fr }\Xi''\star\Theta'')\qquad \{\Xi''\star\Theta'\star\Theta''\}C'''\{\Xi_p\}}{\{\Xi''\star\Theta\star\Theta''\}C';\ C'''\{\Xi_p\}}\,(\text{Seq})}{\{\Xi\star\Theta\star\Theta''\}C;\ C'';\ \epsilon;\ C';\ C'''\{\Xi_p\}}\,(\text{Seq})$$

Figure C.4: TemporalLocality's implementation to be applied after FactorizeFrames

Symbol Index for Chap. 2 to 7

Bibliography

[1] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Tool-supported proof system for multithreaded Java. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, number 2852 in Lecture Notes in Computer Science, pages 1–32. Springer-Verlag, 2003. Cited on page 90.

[2] A. Aiken and A. Nicolau. Optimal loop parallelization. *ACM SIGPLAN Notices*, 23(7):308–317, 1988. Cited on page 170.

[3] G. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991. Cited on pages 6 and 67.

[4] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, pages 1–10. ACM Press, 2000. Cited on page 170.

[5] E. Balland, P. Brauner, R. Kopetz, P-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In F. Baader, editor, *Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. Cited on pages 152, 167, and 171.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. Cited on pages 4, 47, 118, and 135.

[7] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Cited on page 90.

[8] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass — Java with assertions. In K. Havelund and G. Rosu, editors, *Workshop on Runtime Verification*, Electronic Notes in Theoretical Computer Science, 2001. Cited on page 135.

[9] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2007. Cited on page 4.

[10] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *IEEE Computer Security Foundations Symposium*, June 2008. Cited on page 3.

[11] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in

Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006. Cited on pages 152 and 170.

[12] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in Lecture Notes in Computer Science. Springer-Verlag, 2007. Cited on page 64.

[13] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In W. Damm and H. Hermanns, editors, *Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, 2007. Cited on page 170.

[14] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In K. Lodaya and M. Mahajan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 2004. Cited on page 150.

[15] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer-Verlag, 2005. Cited on pages 48, 64, 140, 146, 152, 154, 159, 166, and 171.

[16] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *Asian Programming Languages and Systems Symposium*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, 2005. Cited on pages 150, 152, 154, 155, 159, and 170.

[17] K. Bierhoff. Iterator specification with typestates. In *Specification and Verification of Component-Based Systems*, pages 79–82, 2006. Cited on pages 46 and 47.

[18] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 301–320, 2007. Cited on pages 46 and 47.

[19] A. Bik and D. Gannon. Automatically exploiting implicit parallelism in Java. In *Concurrency: Practice and Experience*, volume 9, pages 579–619, 1997. Cited on page 170.

[20] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Mathematical Foundations of Programming Semantics*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier, 2005. Cited on pages 35 and 169.

[21] R. Bornat, P. W. O'Hearn, C. Calcagno, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages*, pages 259–270. ACM Press, 2005. Cited on pages 6, 20, 46, 138, and 173.

[22] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–230. ACM Press, November 2002. Cited on page 170.

[23] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003. Cited on pages 6, 20, 138, and 173.

[24] J. Boyland. Semantics of fractional permissions with nesting. Technical report, University of Wisconsin at Milwaukee, December 2007. Cited on pages 55 and 174.

[25] J. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *Principles of Programming Languages*, pages 283–295, 2005. Cited on page 47.

[26] J. Boyland, W. Retert, and Y. Zhao. Iterators can be independent "from" their collections. International Workshop on Aliasing, Confinement and Ownership in object-oriented programming, 2007. Cited on pages 46 and 47.

[27] R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. In M. Kaminski and S. Martini, editors, *Computer Science Logic*, pages 323–338. Springer-Verlag, 2008. Cited on pages 149 and 150.

[28] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005. Cited on pages 4 and 118.

[29] C. Calcagno, D. Distefano, P. OHearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*. ACM Press, 2009. To appear. Cited on pages 157 and 170.

[30] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In R. Hariharan, M., and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer-Verlag, 2001. Cited on pages 149 and 150.

[31] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003. Cited on pages 118 and 131.

[32] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, 2007. Cited on pages iii, 44, 117, 125, 128, 135, and 173.

[33] W. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In G. C. Necula and P. Wadler, editors, *Principles of Programming Languages*, pages 87–99. ACM Press, 2008. Cited on pages 47, 137, 150, and 170.

[34] D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001. Cited on page 4.

[35] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005. Cited on pages 47, 118, 135, and 174.

[36] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, March 2004. http://coq.inria.fr/doc/main.html. Cited on pages 7, 140, and 149.

[37] P. Crégut. Personal communication. Cited on page 84.

[38] D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In L. J. Hendren, editor, *International Conference on Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, 2008. Cited on page 170.

[39] F. S. de Boer. A sound and complete shared-variable concurrency model for multi-threaded Java programs. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 252–268, 2007. Cited on page 90.

[40] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490, 2004. Cited on page 44.

[41] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005. Cited on page 4.

[42] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer-Verlag, 2006. Cited on page 170.

[43] D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 43, pages 213–226. ACM Press, 2008. Cited on pages 137, 140, 146, 150, and 170.

[44] M. Dodds, X. Feng, M. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *European Symposium on Programming*, Lecture Notes in Computer Science, pages 363–377. Springer-Verlag, 2009. Cited on page 90.

[45] S. Drossopoulou and S. Eisenbach. The Java type system is sound - probably. In *European Conference on Object-Oriented Programming*. Springer-Verlag, 2007. Cited on page 15.

[46] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Early register deallocation mechanisms using checkpointed register files. In *IEEE Computer*, volume 55, pages 1153 – 1166, 2006. Cited on page 170.

[47] J-C. Filliâtre. Queens on a Chessboard: an Exercise in Program Verification. January 2007. Cited on page 2.

[48] D. Galmiche and D. Mery. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 2008. Cited on page 150.

[49] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. In K. Koskimies, editor, *International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, 1998. Cited on page 152.

[50] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. Cited on pages iii and v.

[51] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In Z. Shao, editor, *Asian Programming Languages and Systems Symposium*, volume 4807 of *Lecture Notes in Computer Science*, pages 19–37. Springer-Verlag, 2007. Cited on pages 64, 65, 68, 87, and 90.

[52] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13):1741–1783, 1999. Cited on page 152.

[53] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, July 2008. Cited on pages 8, 9, 17, 49, 67, and 141.

[54] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. Technical Report 6430, INRIA, January 2008. Cited on pages 15, 28, 34, 44, 46, 54, 55, 64, 92, 95, and 101.

[55] C. Haack and C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(3), 2009. Cited on page 17.

[56] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, 1990. Cited on page 152.

[57] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. Cited on page 3.

[58] A. Hobor, A. Appel, and F.Z. Nardelli. Oracle semantics for concurrent separation logic. In *Programming Languages and Systems: Proceedings of the 17th European Symposium on Programming, ESOP 2008*, number 4960 in Lecture Notes in Computer Science, pages 353–367. Springer-Verlag, 2008. Cited on pages 64, 68, and 90.

[59] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 132–149. Springer-Verlag, 2008. Cited on page 2.

[60] M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. PhD thesis, Computing Science Institute, University of Nijmegen, 2001. Cited on page 15.

[61] M. Huisman and G. Petri. The Java memory model: a formal explanation. In *VAMP 2007: Proceedings of the 1st International Workshop on Verification and Analysis of Multi-Threaded Java-Like Programs*, number ICIS-R07021 in Technical Report. Radboud University Nijmegen, 2007. Cited on page 5.

[62] C. Hurlin. Additional resources (Coq scripts, implementations, examples etc.): `http://www-sop.inria.fr/everest/Clement.Hurlin/papers.html`. Cited on pages 33, 34, 92, 133, 149, and 150.

[63] C. Hurlin. Automatic parallelization and optimization of programs by proof rewriting. In *Static Analysis Symposium*, volume 5673 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag, August 2009. Cited on pages 8 and 151.

[64] C. Hurlin. Specifying and checking protocols of multithreaded classes. In *Symposium on Applied Computing*, pages 587–592. ACM Press, March 2009. Cited on pages 8 and 117.

[65] C. Hurlin, F. Bobot, and A. J. Summers. Size does matter: Two certified abstractions to disprove entailment in intuitionistic and classical separation logic. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*. ACM Press, July 2009. Cited on page 8.

[66] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. Cited on page 15.

[67] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, 2001. Cited on pages 20 and 139.

[68] B. Jacobs and F. Piessens. The verifast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, August 2008. Cited on pages 137 and 140.

[69] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. In *Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, 2006. Cited on pages 64 and 174.

[70] B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *International Conference on Formal Engineering Methods*, pages 420–439, 2006. Cited on page 90.

[71] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *International Conference on Supercomputing*, pages 43–43, New York, NY, USA, 2001. ACM Press. Cited on page 170.

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

[73] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006. Cited on page 16.

[74] G. Krishnaswami. Reasoning about iterators with separation logic. In *Specification and Verification of Component-Based Systems*, pages 83–86, 2006. Cited on pages 46 and 47.

[75] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974. Cited on page 170.

[76] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley, Boston, MA, USA, 1999. Cited on pages 57, 67, 80, and 84.

[77] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *European Symposium on Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear. Cited on pages 64 and 174.

[78] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. Peyton Jones, editors, *Principles of Programming Languages*, pages 42–54. ACM Press, 2006. Cited on page 170.

[79] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992. Cited on page 118.

[80] Sun Microsystems. Java's documentation: `http://java.sun.com/`. Cited on pages 55, 117, and 120.

[81] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Cited on page 4.

[82] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000. Cited on pages 152 and 170.

[83] P. W. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003. Cited on page 156.

[84] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007. Cited on pages iii, v, 5, 7, 35, 64, 67, 68, 69, 71, 154, 156, and 173.

[85] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. Cited on pages 32 and 140.

[86] P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001. Invited paper. Cited on page 35.

[87] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975. Cited on page 5.

[88] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005. Cited on pages iii, v, 7, 15, 19, 20, 33, 35, 47, 65, 129, 167, and 173.

[89] M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005. Cited on pages 46 and 47.

[90] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 75–86. ACM Press, 2008. Cited on page 47.

[91] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Smart Card Research and Advanced Application*. Kluwer Academic Publishing, 2004. Cited on page 44.

[92] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *European Symposium on Programming*, 2009. To appear. Cited on pages 152, 163, and 170.

[93] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*. IEEE Press, July 2002. Cited on pages iii, v, 4, 17, 18, 19, 20, 156, 159, and 173.

[94] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, July 2009. To appear. Cited on page 48.

[95] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, 2008. Cited on page 48.

[96] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical Report UIUCDCS-R-2005-2611, University of Illinois at Urbana-Champaign, 2005. Cited on pages 8, 163, and 165.

[97] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, London, UK, 2002. Springer-Verlag. Cited on page 170.

[98] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *Conference on Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer-Verlag, 2007. Cited on page 90.

[99] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, pages 185–210, 1993. Cited on page 32.

[100] C. Xue, Z. Shao, and E.H.-M. Sha. Maximize parallelism minimize overhead for nested loops via loop striping. *Journal of VLSI Signal Processing Systems*, 47(2):153–167, 2007. Cited on page 170.