# Types for Correct Concurrent API Usage

Nels E. Beckman

CMU-ISR-10-131
December 2010

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Jonathan Aldrich (Chair)
Stephen Brookes
William Scherlis
Sriram Rajamani (Microsoft Research India)

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy.*



This work was supported in part by DARPA grant #HR0011-0710019, NSF grant CCF-0811592, R&D Project Aeminium CMU-PT/SE/0038/2008 in the CMU—Portugal program, Army Research Ofce grant #DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," the Department of Defense, and the Software Industry Center at CMU and its sponsors, especially the Alfred P. Sloan Foundation. The author was supported by a National Science Foundation Graduate Research Fellowship (DGE-0234630).

**Keywords:** API, object protocol, typestate, concurrency, multi-threading, object-oriented, type theory, type system, static analysis, specification, verification, Java, empirical, polymorphism, inference, probabilistic, transactional memory, STM, optimization

*For Dr. Barbara Beckman, a great scientist and a wonderful mother*

# Abstract

This thesis represents an attempt to improve the state of the art in our ability to understand and check object protocols, with a particular emphasis on concurrent programs. Object protocols are the patterns of use imposed on clients of APIs in object-oriented programs. We show through an empirical study of open-source object-oriented programs that object protocols are quite common. We then present "Sync-or-Swim," a methodology and suite of accompanying tools for checking at compile-time that object protocols are used and implemented correctly. This methodology is based upon the existing access permissions method of alias control, which is here extended to be sound in the face of shared-memory concurrency. The analysis is formalized as a type system for an object-oriented calculus, and then proven to be free from false-negatives using a proof of type safety. The type system is extended with parametric polymorphism, or "generics," in order to increase its ability to check commonly occurring patterns. An implementation of the approach, a static analysis for programs written in the Java programming language, is presented. This implementation was used to perform a series of case studies whose goal was to evaluate the ease of use, expressiveness and ability to verify commonly occurring patterns. These case studies are presented. Next, an approach and an associated tool for inferring access permission annotations is presented. This inference tool can reduce the burden of using our protocol-checking approach by automatically inferring the required typing annotations. This inference is built upon a system of probabilistic constraints, which allows the easy encoding of heuristics. Finally, an optimization of software transactional memory runtimes is presented. This optimization is enabled by the typing annotations required to use the concurrent protocol checker and can remove some of the overhead typically associated with transactional memory systems. As a result of the work presented in this thesis, it is possible to guarantee the absence of certain API usage errors even in concurrent programs, and to do so with a low burden on programmers. By adhering to such an approach, programmers can produce more reliable software.

# Acknowledgements

The process of getting a Ph.D. is a long one, and a trying one. During the past five years I have experienced innumerable setbacks. Each one dashed my self-confidence in some small way, and taken together might be considered to be quite discouraging. But through the setbacks I experienced true and powerful transformation, both intellectually and personally. Critically, such a journey would have been unbearable were it not for my amazing colleagues, my wise faculty advisors, and the support of my friends and family.

Carnegie Mellon is an amazing place to study computer science, and it is largely great because of the high quality of its students. I have learned a lot from the students in the Principles of Programming group, our resident programming language theorists. Before arriving at CMU, type theory and formal logic were foreign concepts to me, and without their patient explanations, particularly those of Rob Simmons, William Lovas and Tom Murphy VII, I never would have had a chance to see the beauty that underlies everyday programming.

I was lucky enough to play an advisor role for several talented students, Yoon Phil Kim, Duri Kim, and Paul Richardson. You were all quick learners and exceeded my expectations. Some of your work is described in these chapters! I hope you are all happy with the results. Any errors are, of course, my own.

I am also grateful for all the other software engineering Ph.D. students in the Institute for Software Research. They are an entertaining and intelligent bunch. Make sure you keep the SSSGs well-stocked with treats and lively debate after I am gone. To those students in the ISR who have gone before me, particularly George Fairbanks and Shang-Wen "Owen" Cheng, thanks for all of the useful advice. It was invaluable!

A special thanks goes out to the members of the PLAID research group. The PLAID group is more like a family than a research group. Its members are entertaining and kind, and there is always a lively and contrarian discussion going on amongst its members. I have spent a particularly large amount of time with Thomas LaToza, Joshua Sunshine, Donna Malayeri, Sven Stork and Ciera Jaspan. We have eaten lunch together incalculable times. Sorry for always forgetting to bring money. I will pay you back soon! We have also attended a number of conferences together, and I can say that these have been some of the most fun and rewarding experiences of my career. Thanks a lot guys. Ciera and I started graduate school at the same time, and we have spent a lot of time eating and working together. And now we are going to graduate just about at the same time. Congratulations, and best of luck to you Ciera! I am excited to hear about the progression of your academic career.

An extra bit of thanks goes out to Kevin Bierhoff. In addition to being a good friend, Kevin has been an excellent collaborator. My work has largely continued the work that Kevin started and as such, Kevin and I have worked very closely. Kevin, thanks for being such a great friend and for giving me so many good ideas!

I am very appreciative of my entire thesis committee for generously giving their time to me. All professors are enormously busy, which is why I have been so im-

pressed by the time my committee members, Bill Scherlis, Stephen Brookes, and Sriram Rajamani, have given to meeting with me and reading my work. Their feedback has been invaluable.

While not on my committee, Aditya Nori has been a sort of second advisor to me. My two trips to India to visit Microsoft Research have been transformative, both from an intellectual perspective and from a wider cultural perspective. Aditya has been a big part of these experiences, and since my first trip we have stayed in close contact, sharing ideas and research progress. It was great getting the opportunity to work with him again on a project relevant to my thesis!

And of course I am extremely indebted to Jonathan Aldrich, my thesis advisor and mentor. Jonathan is a great advisor. He is intelligent, naturally, but intelligence for Carnegie Mellon professors is merely par for the course. He is also patient, firm and encouraging, and wields those characteristics as the situation mandates. I am not sure I could have gotten through this process with anyone else. Thank you Jonathan!

When asked to give advice on a successful graduate experience, I have always stressed that one cannot make it through a Ph.D. program without a strong network of friends. This is absolutely true. Fortunately for me, I have a number of great friends and an amazing, supportive family, without whom I never would have made it through this journey.

I have always been grateful for my friends from New Orleans like Rand Voorhies and Nick Perrin, along with my West Coast friends, including Jeff Dralla and Greg Mooney. I am sure you guys have all wondered what I have been up to for the past five years. Well, here you go.

Pittsburgh also has been a great place to meet interesting people. The list of amazing people that I have met here is too numerous to fit in this document, but I would like to especially thank (in no particular order) Emily Keebler, Mike Cronin, Suzanne Weesies, Mike Tschantz, Jason Franklin, Ryan Kelly, Jessica Nelson, Reed Taylor, Laura Halderman, Erika Laing and some of the great friends who have left Pittsburgh like Maja H.Ahmetovic, Laura Hiatt, Stephen Magill, Mike Dinitz, Chris Martin, Marcus Louie and Cortney Higgins. I am all too aware that I have left out someone who has been very kind to me here, and for that I am really sorry!

Thanks to my awesome family for giving me so much support and guidance. My extended family has always been so great. Tom and Holly, Ira and Jay, and Jane, thank-you all. And those who are no longer around to see this day, Jim, Donnis and Ed, I really miss you. Kristin, you are an awesome sister and I am so happy for your success! Mom and Dad, you guys are the best, and you instilled in me the confidence, curiosity and discipline it takes to be successful. Thank you so much.

Finally, Brianne, your love has meant so much to me. You are a kind, intelligent and beautiful woman, and I am so glad we will be spending the rest of our lives together. I will always remember graduate school fondly because, at the very least, it brought us together. I love you!

My advice to future graduate students? Have fun, make friends, fall in love, and learn as much as you possibly can.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

*Let the spirit of adventure set the tone.*

---

## 1.1 Concurrency and Object Protocols

These days it is hard to read much literature in computer science, whether in the popular press or the academic literature, without hearing about multi-core processors and their impact on programming practice. The prevailing wisdom is that, due to the inability of the fastest single-core chips to adequately dissipate heat, those chips will be largely replaced by chips containing numerous power-efficient cores [6]. Anecdotal evidence certainly backs up this assertion! Five years ago this author, along with each member of the incoming graduate class, received a computer with a single-core Pentium IV chip. The next year and every year since incoming students have been issued a multi-core machine. These days one would be hard-pressed to find a CPU on a general purpose computer that does *not* contain multiple cores, with two and four cores being quite common, even on laptops.

The real concern, however, is not manufacturing technology but whether or not we as programmers as a group will be able to write software that will take advantage of computers designed in this manner. Prevailing wisdom says that parallel programming is difficult, a skill obtained through long experience and wielded by experts. If the present hardware trends continue, will we as developers of software, be able to turn parallel programming into a mainstream skill? Or will, perhaps, some new programming methodology make such skills unnecessary?

We do not know what the future of software has in store. But programmers are not waiting around for the programming technologies of tomorrow to make parallel and concurrent programming skills unnecessary. They are writing lots of parallel software today. So the need to ensure quality in such programs is very much a present-day issue. And if it happens new programming paradigms do not replace current parallel programming practices in the near future, quality issues relating to parallelism will only increase in significance.

This trend should be considered in the context of another reality in present-day programming practice, the large-scale reuse of existing code. Today, most "new" applications are only somewhat new, as they will contain large amounts of reused code. This code may be in the form of

a library, for example the large Java standard library, or the reused code may come in the form of a framework, where the new application is more or less "filling in the blanks." In certain domains, for example web applications or IDE tooling, the standard course of action is to *begin* by choosing an off-the-shelf application framework wherein existing code will determine major portions of the program's architecture.

Why is this significant? It means that the correctness of programs now depends in great measure upon the correct use of the libraries and frameworks, generically referred to as application programmer interfaces (APIs), upon which those programs are built. Moreover, correct parallel and concurrent programs will depend upon more that just "simple" correctness properties like a lack of data races and deadlock freedom. They will depend upon correctly using APIs given the additional constraints imposed by the programming paradigm, for example the constraints of thread-sharing.

The Eclipse framework makes for a pretty good example of the present state of affairs, in part because of this author's vast experience using it (see Chapter 3). Eclipse is a modern application framework written in Java. By providing a large, fully-featured application structure with a plug-in architecture, it allows programmers to quickly produce GUI applications by filling in their application-specific details. Eclipse is highly concurrent, running dozens of threads at any given moment, and provides developers with a baffling array of APIs to choose from. Using Eclipse, programmers can release feature-filled tools of an extremely high quality, potentially saving months of years of program development. But doing so requires using its APIs correctly.

This thesis addresses the challenges posed by API use in parallel and concurrent programs. In particular, it focuses on correct use of object protocols in concurrent programs.

Concurrent programs are programs in which the mechanisms of simultaneous operation are visible to programmers as first-class abstractions. For example, most popular languages provide a notion of threads, which abstract numerous independantly executing operations. Threads communicate through a shared memory store, using synchronization primitives like locks and semaphores for coordination. This is worth pointing out because it is different from parallelism, which is the literal simultaneous execution of tasks. Parallelism is what gives us speed-up on multi-core machines, so it is very important. This thesis will focus on shared-memory concurrency, a very important, but by no means the only, paradigm for achieving parallelism.

Object protocols are protocols of correct use that are defined by the types (classes and interfaces) in APIs in object-oriented languages. While all types may be considered to define at least a degenerate protocol, the most interesting protocols are ones in which API clients are required to obey method ordering constraints. For example, consider the abstract class `TrayDialog` shown in Figure 1.1. This class comes from the Eclipse framework, and can be used to add "tray" dialogs, dialogs that can slide into and out of view, to an application. By reading the documentation we can see that this class restricts the order in which two of its methods can be called. In particular, clients must be aware of the current state of the tray; is it open, or is it closed? If it is open, calls to the `openTray` method are prohibited, but the `closeTray` method can be used to close the tray. If it is closed, calls to `closeTray` are prohibited, but the `openTray` method can be used to open the tray.

One of the primary results of this thesis is an approach, a static analysis, that allows us to find violations of just such object protocols before client programs are run by examining the source code of an application. These violations can be found even if the objects in question are shared

```
1  /**
2   * A TrayDialog is a specialized Dialog that can contain a
3   * tray on its side. The tray's content is provided as a DialogTray.
4   * ...
5   */
6  public abstract class TrayDialog extends Dialog {
7    /**
8     * Closes this dialog's tray, disposing its widgets.
9     *
10    * @throws IllegalStateException if the tray was not open
11    */
12   public void closeTray() throws IllegalStateException;
13
14   /**
15    * Constructs the tray's widgets and displays the tray in this
16    * dialog. The dialog's size will be adjusted to accomodate the tray.
17    *
18    * @param tray the tray to show in this dialog
19    * @throws IllegalStateException if the dialog already
20    *         has a tray open
21    * @throws UnsupportedOperationException if the dialog does not
22    *         support trays, for example if it uses a custom layout.
23    */
24   public void openTray(DialogTray tray)
25      throws IllegalStateException, UnsupportedOperationException;
26
27   // Class continues...
28 }
```

Figure 1.1: The `TrayDialog` class which defines a simple protocol relating to the "tray" metaphor

amongst multiple program threads.

## 1.2 Challenges

Our goal is to help developers use APIs correctly in concurrent programs by detecting and reporting violations statically. There are a number of challenges that make our goal more difficult. One of the first design decisions that must be made is whether to use a global analysis or a local analysis. Global analyses have the benefit that they may be more convenient to use, since they generally require little or no assistance from the programmer. Unfortunately, global analyses typically do not scale to large sized programs due to their resource requirements.

Since scalability is a desired quality attribute of our approach, a modular, intra-procedural approach is a better starting point. Specifically, this approach is type-based, influenced by the convenience and strong guarantees of popular statically typed languages such as Java and Standard ML. Part of the benefit is familiarity. Many programmers have become accustomed to using such strongly typed languages and to the programming process to which they lend themselves. In

such languages program modules are type-checked one at a time. The type-checker never needs to return to a module that was type-checked previously. Modules do not need to be rechecked when used in different contexts or under different assumptions, as might be the case in a global analysis. This behavior is enabled by typing annotations that form specifications for program boundaries. This in turn allows a development process that is isolated or component-ized. Developers can work more or less independently, using the module specification, which in our case would contain a description of the required protocol, as a stable standard against which they will program.

The idea of checking protocols as types is not new. Strom and Yemini [90] proposed the idea of modeling certain program properties as finite state machines and checking them statically in a methodology known as "typestate." But type-checking protocols also brings along a suite of challenges.

Most of the challenges of concurrent protocol type-checking can be attributed to soundness. Most type systems for programming languages are *sound*, meaning broadly that the languages respect the abstractions they define. For example, in Java a programmer cannot define a class as a series of fields of object type and then treat instances of that class as a flat array of bytes. Allowing such a thing could lead to situations where, after the bytes in the array had been modified, other clients of the class read what they believe is an object but is in fact a meaningless sequence of bytes.

We would like our type-based protocol checker to be sound, meaning that if the type-checker says a protocol is not violated than it will never be violated. This brings with it a number of challenges.

The first challenge is aliasing. When multiple variables in a program point to the same object at run-time, those variables are said to be aliased. If an analysis is modular and cannot examine every portion of the program whenever it deems fit, then it must have some idea about whether or not variables in a module might alias or else it must be painfully conservative. Existing approaches have solved this problem using alias-control mechanisms such as ownership [30, 31] or access permissions [15]. We will use access permissions to get a better handle on aliasing in our approach.

The next challenge is thread-sharing. How can a modular analysis, which is local, understand the thread-sharing patterns of a program, when such a feature is inherently global, architectural concern? Such information is needed so that we can tell when an object that must obey a protocol might be concurrently modified by another thread. Generally the answer would be through new annotations, such as in a number of type-based race detection systems [24, 43, 85]. In our approach, our means of alias control, access permissions, doubles as our means of thread-sharing control.

Our solutions to the previous challenges, however, lead us to a new challenge, that of specification burden. As the types of program variables begin to take on more and more complex information (e.g., aliasing, thread-sharing, object state), it becomes more and more of a burden for programmers to write down such types as module boundaries. Can a type system still be usable if its types encode so much information?

## 1.3 Overview: To Sync, or to Swim?

Our approach, which will will broadly refer to as the "Sync-or-Swim Methodology," solves many of the aforementioned challenges, while at the same time establishing that protocol conformance is an important area of study.

### 1.3.1 Do Protocols Really Matter?

Our first step is to better motivate the need for static protocol checking by presenting the results of an empirical study on object protocols. In most of the existing work on typestate checkers and protocol conformance readers will notice the startling similarity between the example protocols discussed. Generally such papers will start off by describing a simple file or socket data structure with open and closed states. Given the prevalence and similarity of such examples, one is lead to wonder, are files and sockets the only data structures that define such protocols? If this were the case, then it would be harder to argue that the effort spent attempting to verify protocol usage is well-motivated.

It was our hypothesis, however, that object protocols were both more common and more varied than implied by such simple examples. We set out to test this hypothesis by developing a simple static analysis to detect protocol definition and running this analysis on a number of popular open-source applications and libraries. The results showed that a great deal of the types in these programs defined object protocols, and an even larger number used them, suggesting that protocol conformance is in fact worthy of study.

This work is presented in Chapter 2.

### 1.3.2 Checking Protocols the Sync-or-Swim Way

The bulk of the thesis is spent explaining our approach for statically, modularly and soundly checking protocol conformance in concurrent applications. We call our approach the "Sync-or-Swim Methodology," after the Sync-or-Swim static analysis tool which embodies the approach.

The power of the analysis is largely derived from access permissions, an alias and concurrency control methodology first developed by Boyland [25] and extended by Bierhoff and Aldrich [15]. In the Sync-or-Swim methodology, types like File or Socket are extended with access permissions, a flow-sensitive type qualifier that both describes the current abstract state (e.g., open/-closed) of the reference and succinctly describes the ways in which the referenced object may be aliased or thread-shared.

For example, consider the code excerpt in Figure 1.2. In this piece of code the `TrayDialog` class, presented in Figure 1.1, is used by a client in a call-back method. This example is particularly simple, but more interesting examples will be explored over the course of this thesis. The type of the parameter `td` is enhanced with an access permission, the `@Unique` annotation. This access permission says that `td` will point to an unaliased, thread-local object, and that the tray will be open when the callback method is invoked. Thanks to this information, verification of the method can proceed simply. The call to the `closeTray` method, which requires the tray to be open, is guaranteed to succeed because `td` is guaranteed to be open and we are ensured that no other thread could be modifying the tray concurrently.

```
1 void closeTrayCallback(@Unique(requires="Open") TrayDialog td) {
2   // ...
3   td.closeTray();
4 }
```

Figure 1.2: A client of the `TrayDialog` class. Thanks to access permissions, which enhance the notion of type, its correct use of the tray's protocol can be verified.

Our approach is formalized as a type system but implemented as a static analysis for concurrent Java programs. This implementation is known as "Sync-or-Swim," for the correct thread synchronization it helps programmers employ. Although given the metaphor, and because of the problems that can occur due to improper synchronization, it might be better known as "Sync-or-Drown!"

One contribution of our approach is the reinterpretation of access permissions as thread-sharing permissions. One of the benefits of this reuse is that, thanks to an imposed synchronization discipline, a concurrent program verified in our approach requires no additional specifications over and beyond what is required for a single-threaded program. This is potentially an important boon to developer productivity given the challenges of specification burden. This leads us to our next point.

### 1.3.3 One More Developer Responsibility?

In order to minimize programmer burden associated with correct specification we have also developed a specification inference technique. While the benefits associated with static checking of object protocols may be significant, because we have chosen to create a modular, specification-dependent analysis, there is a risk that the benefits of static checking may be outweighed by the extra annotation work required of programmers.

In order to lessen this burden we have developed Anek, a global, *probabilistic* tool for inferring access permission specifications. Given an annotated API, Anek will add to a client program the specifications necessary to statically check protocol conformance. Anek is interesting in part because it attempts to encode developer intuitions into its analysis. For a given program, Anek will generate a series of probabilistic constraints over Boolean random variables. Each one says that a specification at some program point is *likely* with a particular probability. When these constraints are solved, the most likely specification is chosen. The probabilities themselves come from our own experiences writing access permission specifications, plus additional consistency constraints. They encode which specifications are likely to be good in particular program scenarios. One of the benefits of this approach is that, even in the face of program bugs or analysis imprecisions, specifications that are "good" in some sense can still be inferred.

This work is presented in Chapter 7.

## 1.4 This Thesis

This work evaluates the thesis statement presented in this section, along with a number of smaller, related hypotheses.

### 1.4.1 Thesis Statement

Access permissions, which statically describe the aliasing behavior of program references in object-oriented programs, provide a good basis for the lightweight verification of object protocols in concurrent systems, allowing us to verify real programs and provide optimizations of certain runtime systems.

### 1.4.2 Hypotheses

We can break the thesis statement down into more concrete, and measurable hypotheses.

#### Hypothesis 1: Prevalence of Object Protocols

Object protocols are an important and recurrent pattern in object-oriented development, and therefore are worthy of further study.

**Validation** In order to validate this hypothesis I have completed a large study of open-source Java software in order to determine the nature and frequency of object protocols as they occur, "in the wild."

#### Hypothesis 2: Formalization

We can develop and formalize an analysis that will guarantee a concurrent program does not violate the object protocols that it defines and prove that the system will not produce false negatives.

**Validation** This hypothesis will be validated by developing and formalizing a type system and operational semantics based on our permission system and proving the type system sound with respect to its semantics. The proof essentially says that no object in a program will ever be required to be in some abstract state that at run-time it will not actually be in.

#### Hypothesis 3: Specification Coverage

Our specification system can be used to specify the behavior and implementation of object protocols in real concurrent, object-oriented programs.

**Validation** In order to validate this hypothesis, I have specified the behavior of object protocols in numerous small and two large concurrent Java programs collected from open-source projects. During this process I noted and here report the recurring and interesting patterns of protocols that can and cannot be specified.

**Hypothesis 4: Analysis Precision**

Our analysis will report a relatively low number of false positives, on the order of the number of false-positives reported by comparable automated behavioral analyses.

**Validation** In order to validate this hypothesis, I have built an automated static analysis for Java and used it to check the specifications on the suite of case studies. The rate of false positives per line of source is reported herein and compared with other approaches.

**Hypothesis 5: Mutual Exclusion Requirements**

In order for a program to be verified, it should not require a great deal more critical sections than is strictly necessary for functional correctness.

**Validation** During the verification process I have observed and reported on the number of times that my analysis forced me to add synchronization in the cases where the original programs were synchronized correctly.

**Hypothesis 6: Probabilistic Inference**

Probabilistic specification inference, an inference that can encode intuitions describing common or "good" specifications, is a good solution for reducing programmer annotation burden, resulting in specifications with comparable false-positive rates as those written by hand.

**Validation** In order to evaluate this hypothesis we have created a probabilistic inference tool Anek, and evaluated its performance inferring specifications for a large open-source program. The specifications inferred were compared with ones written by hand for both subjective quality and the number of false positives they gave rise to when subsequently running our protocol checker.

**Hypothesis 7: Optimization**

Because access permissions describe aliasing behaviors, permission annotations can be used to optimize transactional memory runtimes, improving their performance.

**Validation** In order to validate this hypothesis, I have modified a source-to-source implementation of transactional memory for Java to remove unnecessary synchronization and logging based on the access permission annotations. I have compared performance of this optimization by running a benchmark suite with and without the change.

## 1.5 Contributions

This thesis shows that access permissions, a simple but flexible alias-control mechanism can be used to soundly verify protocols in concurrent programs. This thesis makes the following contributions:

1. In order to better motivate our work on verifying object protocols, this thesis *establishes the frequency* of object protocol definition and use in Java programs, and provides a taxonomy of those protocols.
2. We show that a particularly flexible form of access permissions, previously presented in the context of single-threaded protocol conformance, can be used to *guarantee the absence* of protocol errors in concurrent object-oriented programs. When compared to existing systems for verifying non-trivial properties of concurrent programs, our permissions are either more flexible in the types of aliasing or thread-sharing they allow, or can be checked in an automated fashion where others cannot.
3. This system of access permissions is then made more expressive by the addition of *bounded parametric polymorphism*, a first for similar styles of alias-control mechanism, necessary in part because of flexibilty of our aliasing permissions.
4. We establish the *practicability* of our approach by evaluating our static analysis on open-source programs written by other developers. The total amount of code verified is large as is the number of APIs whose correct use was verified. During this evaluation process, we show that the burden on developers is relatively low, as is the rate of false positives, which includes the number of times programmers are forced to add unnecessary and potentially costly thread synchronization.
5. We establish that probabilistic constraints are a good foundation for the *inference of modular behavioral specifications* by developing such an inference and then showing that the specifications it generates are of a similar quality as those written by hand.
6. Finally, we establish that modular aliasing specifications (access permissions, and others like them) are a *good source of information* to feed into concurrent optimizations.

While the general theme of this thesis is strongly related to object protocols in concurrent programs, some parts describe work that is not strictly related to concurrency. As a case in point, our empirical study does make mention of concurrency primitives in certain cases, but more broadly could be used to motivate all forms of protocol checking. Our polymorphic extension likewise is an extension to a single-threaded protocol checker, as the interesting features of this work are largely orthogonal to concurrency. Still, polymorphism was motivated by our concurrent case studies, since many of the multi-threaded applications we examined made use of generic work queues. Finally, while our system of inference would work for concurrent programs, it was designed and evaluated with single-threaded programs in mind. Still, all of these features would be desirable while checking concurrent programs, even if they do not directly relate to concurrency per se.

## 1.6 Potential Impact

This work continues a recent trend of exciting work that is putting static concurrency checking into the hands of practicing programmers. Our continued focus on *atomicity* for preserving program invariants will help to propagate the view that correct concurrency is about more than the simple avoidance of data races and deadlocks. This work also contributes an in-depth discussion of "check-then-act" races conditions that are often due to clients using object protocols on thread-shared objects. Such a focus provides a better framework for discussing and understanding this class of defect.

More concretely, this work provides an approach and a software engineering tool that makes correct concurrency nearly as convenient as type-checking. Developers need only decide which protocols in their program are intricate enough to warrant static checking, specify the protocol and the aliasing specifications for those objects and correct concurrency will be guaranteed.

The author expects that the taxonomy of object protocols will become especially useful, both for practicing programmers and for researchers. Programmers who are looking for a better understanding of protocols as they occur in practice can study the taxonomy. Afterward, such a programmer will have a good appreciation for all of the different protocols they can expect to see. For future researchers developing protocol-checking tools, this taxonomy can provide a litmus test of sorts. Researchers whose approach is expressive enough to check (and specify, if appropriate) each category of protocol should be reasonably confident that their approach will work for most software. We also believe that this work will lead to more empirical investigation into the nature and commonality of object protocols.

Some of the results presented in this work have already inspired interesting follow-on work. For example, when an analysis has enough information to determine the dependencies between different threads in a program (information that our access permissions provide), might that analysis then be able to automatically parallelize the program, rather than simply checking verifying correct concurrency? Stork et al. [89] have already begun to investigate this possibility with their Æminum language. Automatic parallelization has long been a holy grail for programming researchers, but in the past performance gains were modest because their tools did not have the sort of static sharing information given by our access permissions. Now big advances appear to be possible.

Recently Aldrich et al. [4] have proposed "typestate-oriented programming." This paradigm as a foundation for general-purpose programming was at least partially inspired by the ease of type-based protocol checking, as embodied by the Sync-or-Swim tool, and the prevalence of object protocols in the real world, as observed by our empirical study. While the Plaid language, the embodiment of typestate-oriented programming, currently has no support for parametric polymorphism, it is certainly planned for the future in order to support generic permissions to objects contained in data structures. Any system of parametric polymorphism for this language would likely end up being very similar to the polymorphic permission system presented in this paper. Considering these developments, it seems that the future of access permission and concurrent object protocol checking will be an interesting one!

## 1.7 Thesis Outline

This thesis contains seven chapters besides the introduction and conclusion. It proceeds in the following manner:

Our initial hypothesis was that access permissions would be a solid foundation for verifying object protocols in concurrent programs. What we did not know, however, was how well-motivated such verification was. To this end, Chapter 2 contains an empirical study of both the definition and use of object protocols in open-source Java programs. The end result is that protocols were found to be quite common. A taxonomy of the protocols we encountered is also presented. The next chapter, Chapter 3, presents our approach, a type system for checking that protocols are used correctly in concurrent programs. Examples of protocols and their verification are presented and we discuss Sync-or-Swim, the static analysis for Java programs that embodies the principles of this type system. Chapter 4 describes the soundness guarantees provided by this type system. It contains a proof of type safety for a core language that contains the most important features of our approach. Readers not interested in the full details of the proof can read the first section of this chapter, which contains a summary. The complete proof continues into Section B of the Appendix.

During our early case studies we encountered various programs that our initial approach was unable to verify. While some of these problems were due to trivial issues in the theory or implementation, it was determined that by adding parametric polymorphism to our type system, it could be made much more expressive with little additional annotation burden. Our system of polymorphic permissions, along with some of the examples that motivated it, is presented in Chapter 5. Chapter 6 presents the evaluation of the complete system. In the evaluation we used Sync-or-Swim, the implementation of our complete approach, to examine the expressiveness, precision and burden of our methodology.

Next, in Chapter 7, because of the potential for a large specification burden we present Anek, a tool that can infer specifications for Sync-or-Swim, and its methodology, which is built upon probabilistic constraints. We evaluate its performance as well. Chapter 8 presents an optimization of software transactional memory based on the permission specifications presented in this thesis. These optimizations, which help make the "atomic block" mutual exclusion primitive more practical, helps to highlight the potential for mutual benefit when information-rich types such as ours are used for program verification. An evaluation is presented. Finally, in conclusion, Chapter 9 revisits our hypotheses.

# Chapter 2

# An Empirical Study of Object Protocols in the Wild

*Learn to listen, not hear.*

---

This chapter presents an empirical study of object protocols in several large open-source applications, including the Java standard library. The goal of this study is to better understand how protocols are defined and used in large programs, and to help better motivate the remainder of the thesis along with the large amount of existing research that has been done on object protocol conformance. This work was done in collaboration with Duri Kim, and forms the basis for her masters thesis [68].

## 2.1 Introduction

Object protocols are rules dictating the ordering of method calls on objects of a particular class. We say that a type defines an *object protocol* if its concrete state can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition (a definition we will expand in Section 2.2.1). The classic example of an object protocol, often cited in research literature, is that of a file class. Instances of this file class can only have their `read` methods called while the file is open. Once the file is closed with the `close` method, subsequent calls to the `read` method will result in run-time exceptions or undefined behavior. Most popular languages do not give object protocols first-class status, and therefore cannot ensure their correct use statically.

Static and dynamic checking of object protocols is an extremely active area of research in the software engineering and programming languages communities. (Some protocol checking tools are known as "typestate checkers," a more commonly-heard term.) There have been protocol checkers based on software model checking [7, 42]. There have been type systems and flow analyses for checking object protocols [15, 30, 71, 90]. There have been checkers that focus on the narrower problem of object initialization [40, 82], and checkers that focus on the wider issues of framework conformance [41, 65]. There have even been dynamic checkers [14, 49],

and checkers that focus on concurrent applications [10, 67].

While many of these approaches are quite powerful and their designs elegant, we argue that very little is known about how protocols are used in practice. Are protocols a big problem, or a niche issue? Do they occur often or are they rarely defined? Are they used by many other classes? Are the protocols themselves simple, or complex? These are the kinds of questions we have attempted to answer with this study.

In this chapter, we present an empirical study on object protocols in open source Java software. We took several popular open-source projects and the Java standard library, ran a suite of automated analyses that attempted to find evidence of object protocols, and manually investigated the results of those analyses.

This work contains several contributions. As part of our investigation, we discovered that object protocol definition is relatively common (in about 7% of all types) and protocol use even more so (by about 13% of all classes). We discovered seven behavioral categories of object protocols that account for 98% of all the protocols we discovered. Finally, the methodology itself is somewhat novel, in that we used a very simple static analysis to identify a pattern indicative of object protocols. This dramatically reduced the amount of code we needed to examine manually.

The chapter proceeds in the following manner: Section 2.2 discusses the design of our experiment. This includes important definitions, description of our automated analyses, the data that we gathered and the motivation underlying our approach. Section 2.2.4 describes the threats to the validity of our experiment. Section 2.3 presents the data that we gathered during our study, and Section 2.4 discusses that data and its implications for other researchers.

## 2.2 Methodology

Our study proceeded in the following manner. We created a static analysis to detect patterns in source code that we believe are indicative of object protocols. Then, we ran the static analysis on popular open-source Java projects and the Java standard library. Next, we manually investigated the reports issued by the static analysis, marking each as evidence for a protocol or not. During this process, data about the location, classes involved, their super-types, and more was gathered. We also created categories of similar protocols based on our observations. Finally, we used the information about which types define protocols in order to run another automated analysis which gathered information about the usage of those protocols.

The first part of this section will discuss our definitions, namely, what are object protocols? The next part walks the reader through the experimental process, including a description of our analyses and the data we gathered. Finally, we describe the Java programs we analyzed and threats to the validity of our study.

### 2.2.1 Definitions and Scope

One of the trickiest parts of discussing object protocols is agreeing on exactly what is meant by the term. While many sanctioned interactions between different pieces of code could be described generically using the term protocol, we choose to focus on a definition that is based around abstract state machines. The definition of object protocol stated here sets the scope for

our entire experiment. It is the idea on which our analyses and terms like "false negative" will be based.

**Definition** A type defines an *object protocol* if its concrete state can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition.

This definition contains several key ideas.

**client** The states of the protocol must be observable and relevant to clients.

**abstract and finite** The states must be abstractions of any internal representation, and there must be a finite number.

**runtime transitions** Methods calls on an object instance after construction will cause it to transition between abstract states.

**correct use** Failure of clients to obey a protocol can result in run-time exceptions or undefined behavior.

Classic examples of protocols fall under this definition. For example, an instance of the `java.io.FileReader` class can be interpreted as having two abstract states, Open and Closed. Clients must be aware of which state a given instance of the file is in otherwise they might incorrectly call a method such as `read`, which requires the file to be open, when the file is actually closed. `java.util.Iterator` fits our definition as well. Even though it is an interface and does not have its own concrete state, clients must be aware that the `next` method can only be called when a call to `hasNext` would return true.

Our definition includes initialization protocols; objects that must have certain methods called after construction to put them into a valid, *initialized* state. While these protocols may in fact be quite simple, they fit our definition, and are an important piece of the contract of many types.

We additionally include a degenerate form of protocol known as type qualifiers [38, 45]. In this case, object instances enter an abstract state at construction-time that they can never leave. Like other protocols, depending on the state the object is in, certain method calls may be illegal. We will point out type qualifiers in this study even though they do not strictly fit our definition, as we feel they are quite similar to more standard object protocols and because, like object protocols, current languages do not check them statically.

Our definition specifically excludes protocols in which a type has an infinite number of abstract states. This is meant to exclude types such as `java.util.List` on the basis of methods like `List.remove(int)`. This method throws an exception when the argument is greater than or equal to the size of the list. While `List` could be interpreted as having the abstract states, LargerThan0, LargerThan1, LargerThan2, etc., this does not fall under our definition, and will not be considered a protocol.

**Scope of this Study** Our definition of object protocol leaves out other object protocols that some readers may consider to be important. For example, it does not include multi-object protocols, in which clients must call an ordered sequence of methods on two or more objects. One of the things we will show is that, even when taking a restricted view of object protocols, they are still rather common. By considering a more inclusive definition, we believe one would find that object protocols are even more common.

We have observed that protocol classes frequently are implemented so that they can detect protocol violations. Generally, violations that are detected will cause an exception to be thrown (e.g., `InvalidStateException`). This is relevant to our study because, within the scope of our definition of object protocol, our automated analysis detects the *subset* for which this is true (see Section 2.2.2).

**Other Definitions** Here are some other terms that will be used throughout the remainder of the chapter:

**Phase 1** In the first phase of the study we examined the nature of protocol definition.

**Phase 2** In the second phase of the study we examined protocol use.

**Candidate, Candidate Code** A section of code that may represent evidence of an object protocol, as reported by our static analysis.

**Protocol Evidence** A candidate that, after manual analysis, is determined to be evidence of an object protocol (a true positive).

**Evidence Class** A class that contains protocol evidence.

### 2.2.2 Experimental Procedure

Our experiment consisted of several steps where we alternatingly performed analyses, manual and automated, and gathered and processed their results. This section presents the entire process from start to finish. For convenience, this process is illustrated in Figure 2.1. At each step in the experiment, we will say what data is gathered and why that particular course of action was chosen.

#### Phase 1: Finding Object Protocols

In the first phase of our experiment, we start with a set of programs in which we would like to find object protocols. The first step is to run ProtocolFinder, a static analysis that will generate a list of code candidates, locations in code that *may* indicate that a class is defining an object protocol.

We had several goals in mind when developing the ProtocolFinder static analysis. For one, we wanted to keep the rate of false negatives as low as possible. In this case, false negatives are protocols that exist in the programs under analysis that are not found. Manual inspection, of course, can have a very low rate of false negatives but is extremely time consuming, particularly considering the amount of code we would like to investigate. We desired an automated analysis. Dynamic analyses for discovering protocols in running programs exist [60, 101]. Unfortunately, such approaches are quite susceptible to false negatives, since appropriate test cases must be found to exercise all of the possible protocols in an application. For the same reasons, a dynamic approach would require examining only programs that were accompanied by sufficient test cases, and thus, was ruled out. By comparison, a static analysis can be run on any open-source program. In the end, we decided to develop a conservative static analysis that would eliminate many (although not all) false negatives while reducing manual effort. A subsequent manual examination of the results would be used to eliminate any false positives.

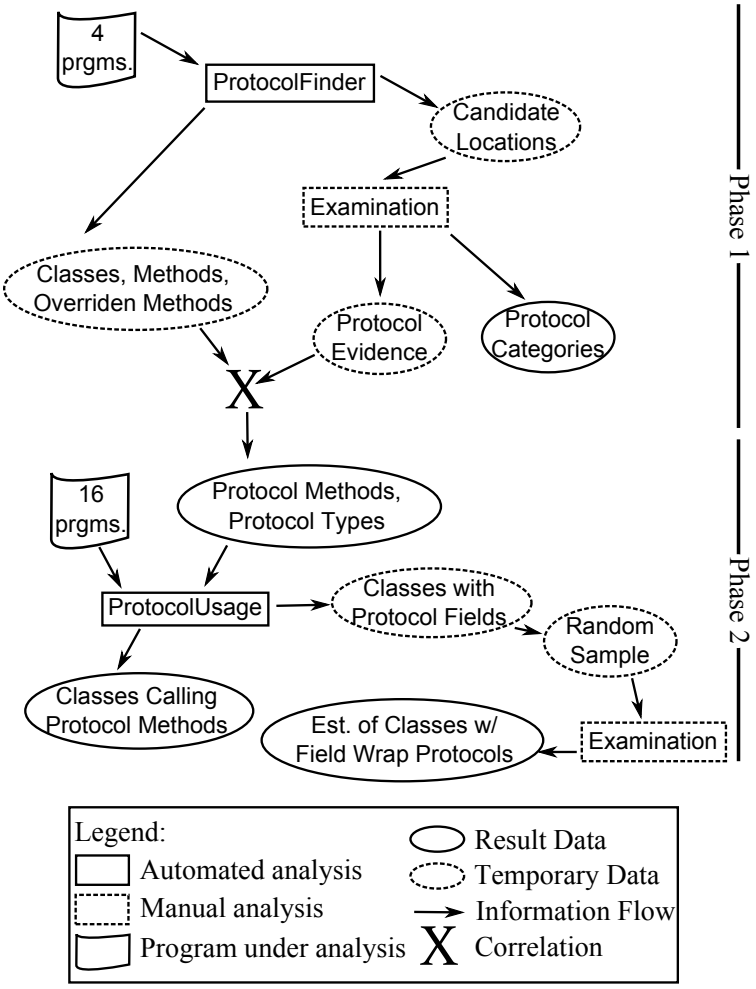


Figure 2.1: A schematic explaining the experimental procedure

ProtocolFinder is a static analysis created for this study that attempts to find object protocols by searching for locations in code where protocol violations are detected. Specifically, it looks for locations in code where instance methods throw exceptions as a result of reading instance fields.

The intuition behind the analysis is simple: In our earlier case studies we noticed that many protocol methods throw exceptions when object protocols are violated. Because our definition of object protocol depends on some abstract state of the method *receiver*, we expect that any exceptions thrown for protocol violation will be thrown in instance methods and as a result of reading an instance field. This pattern has been noted and used as the basis for existing protocol detectors [5, 97].

Like ProtocolUsage, described later, the ProtocolFinder analysis is an Eclipse plugin whose source we have made freely available.[1]

[1] `http://code.google.com/p/nolacoaster/`

```
1  // from java.util.concurrent.ArrayBlockingQueue.Itr
2  public void remove() {
3    final ReentrantLock lock = ArrayBlockingQueue.this.lock;
4    lock.lock();
5    try {
6      int i = this.lastRet;
7      if (i == -1)
8        throw new IllegalStateException();
9      lastRet = -1;
10     // ... method continues
11 }
12
13 // from javax.swing.undo.AbstractUndoableEdit
14 public void undo() throws CannotUndoException {
15   if (!canUndo()) {
16     throw new CannotUndoException();
17   }
18   hasBeenDone = false;
19 }
20
21 public boolean canUndo() {
22   return alive && hasBeenDone;
23 }
```

Figure 2.2: The ProtocolFinder reports candidate code on lines 8 and 16. Both are classified as protocol evidence. In the first, the field `lastRet` flows through a local variable `i`. In the second, the field value comes from a getter.

**ProtocolFinder** ProtocolFinder is a flow-insensitive static analysis that examines every instance method in a given code base. Upon encountering an 'if' block or a conditional expression, the analysis first examines the condition. If the condition expression contains a read of a field of the current receiver (or a call to a "getter" method on the current receiver), the analysis will examine both 'then' and 'else' branches. ("Getter" methods are methods which more or less immediately return the value of a field.) If either branch of the conditional throws an exception the analysis issues a report indicating that piece of code is a protocol candidate. Both the field read in the condition and the throw statement in the branches can be nested arbitrarily deeply. In order to determine whether an expression in the condition is a field read or getter call on the current receiver, the analysis queries a sub-analysis. This flow-sensitive static analysis has a list of all the methods in the current class determined to be field getters, and can track if a value in an intermediate variable flows from a getter or a field.

The analysis uses a simple procedure to determine which methods are "getter" methods. Any method with a non-void return type for which all return statements contain field reads or values that flow from field reads are marked as getters.

ProtocolFinder reports protocol candidates in the examples shown in Figures 2.2 and 2.3, all of which are from the Java standard library. In Figure 2.2, reports are issued on lines 8 and 16. The first example comes from an implementation of the `Iterator` interface. It is noteworthy because the field value flows from the `lastRet` field to the local variable `i` before the conditional.

```
1 // from java.awt.Container
2 public void remove(int index) {
3   synchronized (getTreeLock()) {
4     if(index<0||index>=this.component.size()){
5       throw new ArrayIndexOutOfBoundsException(index);
6     }
7   // ... method continues
8 }
```

Figure 2.3: In the remove method the ProtocolFinder reports a possible protocol on line 5. Note that the field, component, is nested in a sub-expression of the condition (line 4). By manual examination, we have determined that this candidate is *not* evidence for an object protocol.

The second example is noteworthy because the condition involves a call to the getter method `canUndo`, which itself is the result of a combination of fields, `alive` and `hasBeenDone`.

In Figure 2.3, ProtocolFinder reports a candidate on line 5. This example is noteworthy because the field read that occurs on line 4 is nested within a sub-expression of the condition. ProtocolFinder still treats the condition as being dependent on a receiver field.

The output of the ProtocolFinder is thus a list of protocol *candidates*. In the next part of the experiment, we manually inspect each candidate to determine whether or not it is actually evidence of an object protocol. For each report issued, the ProtocolFinder includes the line number and file name of the candidate, the method and class in which the candidate was found, and all methods that are overridden by the method in which the candidate was found. This information helps us find the candidate for the purposes of manual examination, and, in the event that a candidate represents evidence of an actual protocol, will provide us with the data we need to carry out the usage phase of our study.

**Manual Examination** After running the ProtocolFinder and gathering a list of protocol candidates, we investigated each candidate by hand. The primary purpose of this manual investigation was to determine which candidates were actual evidence of object protocols and which were not. This was done by looking at the code location and the surrounding class and trying to understand its behavior. Where possible, documentation was also examined. After understanding the candidate and the conditions under which an exception would be thrown, we consulted our own definition of object protocol in order to determine whether or not the candidate represented protocol evidence.

As an example, consider the code snippets in Figures 2.2 and 2.3. Both were returned as candidates by the ProtocolFinder. During manual analysis, both candidates in Figure 2.2 were classified as evidence for actual protocols. Iterators have *RemovalPermitted* and *RemovalNotPermitted* abstract states, transitioned to and from by the `next` and `remove` methods. `remove` can only be called on instances in the *RemovalPermitted* state. `AbstractUndoableEdit` defines several abstract states but the `undo` method can only be called if an instance is both *Alive* and *HasBeenDone*. The candidate in Figure 2.3, on the other hand, was not categorized as protocol evidence. The exception is really thrown in response to the state of the argument, not the receiver. Even if we wanted to abstract the concrete state of the receiver to prevent the exception, the only

reasonable abstraction would require an infinite number of states.

For every candidate that is manually classified as evidence of a protocol, certain information is recorded and used in the second phase of the study. For each piece of protocol evidence, we record the method in which it appears and the class in which that method appears. These classes are referred to as *evidence classes*. But we consider a larger set of types to be protocol-defining. The methods in which protocol evidence appears, and every method they override or implement are considered to be *protocol methods*. Additionally, any public method that calls a private protocol method is considered to be a protocol method (*if* we determine the private method to be part of the "state check" pattern, described below). Finally, the types declaring each of the protocol methods are known as *protocol types*. When we say in the introduction that 7.2% of types declare protocols, these are the types that we are referring to. As these terms will be used frequently in the rest of the chapter, we summarize:

**Protocol Methods** The methods containing protocol evidence, any methods they override and, if a method containing protocol evidence is private, any public method that calls it.

**Protocol Types** The classes and interfaces containing protocol methods.

Our inclusion of private methods and overridden methods is worth further discussion. Regarding our inclusion of overridden methods, our logic here is that, because of subsumption, any subtype may be known statically as its supertype. When a subtype method is part of an object protocol, overridden methods are also frequently part of a protocol, or at best clients must be aware that some subtypes have usage protocols. Therefore, we want to consider calls to those overridden methods as potential client-side uses of protocols.

This strategy addresses one limitation of the ProtocolFinder, that it cannot detect Java interfaces that define a protocol. If the implementing methods of an interface have behavior that the ProtocolFinder recognizes as a protocol, the interface methods will be added to our list of protocol methods, because they are overridden.

In a few cases, we removed overridden methods that were added to the set of protocol methods by this process because we felt that the methods are widely used and not normally considered to be part of a protocol. For example, in the Java implementation of the Kerberos authentication protocol, the `KerberosTicket` class defines a protocol of which its `toString` method participates; if a Kerberos ticket has been destroyed, calling its `toString` method results in an `IllegalStateException`. However, `Object.toString` should not be considered a protocol method since most implementations do not have such behavior, and it is so widely used that considering it to be one would result in vastly distorted results. (In such situations, one may reasonably conclude that behavioral subtyping was broken.) Figure 2.4 contains the full list of supertype methods that were removed from the list of protocol methods because they do not in general represent protocols. As far as we can tell, no other widely used supertype methods were misclassified in this manner.

We included the public callers of private methods because we noticed a common pattern in many classes we encountered. Private methods cannot be called outside of the class in which they are defined and as a result will never appear as client usage in the second phase of our study. However, many classes have private "state check" methods which verify that the instance is in some particular state. These methods are called by multiple public protocol methods as a way of avoiding code duplication. For example, the `java.util.PrintStream` class defines

```
java.lang.Runnable.run()
java.lang.Thread.run()
java.lang.Object.toString()
java.util.List.add(int,Object)
java.util.List.remove(int)
java.util.AbstractList.add(int,Object)
java.util.AbstractList.remove(int)
```

Figure 2.4: Superclass and interface methods automatically considered to be protocol methods due to a subclass that we *removed* from our list of protocol methods. This was done because these methods are widely used, but their contracts do not imply a protocol.

a simple Open/Closed protocol, and once the stream has been closed, there are essentially no methods that can be called on the stream. In order to implement this without code duplication, the `PrintStream` method defines a private `ensureOpen` method that is called first thing inside every public method of the class. We want to make sure that we consider those public methods to be protocol methods, even though our analysis does not report them, so we add them when our manual analysis confirms this pattern.

During manual analysis of protocol candidates, two final pieces of data are generated. One of the goals of our study is to determine if object protocols share similar characteristics. Anecdotally, most protocols seem to be rather simple, and somewhat similar (e.g., Open/Closed, Initialized/Uninitialized) and we wanted to determine if this was generally true. While manually examining each potential protocol, we did our best to observe similarities and group them into categories based on these similarities. Rather than defining the categories a priori, we constructed them as new similarities were observed.

Lastly, for the purposes of this thesis we are very interested in whether or not protocols are used in multi-threaded applications. We would like to understand the relevance of protocol checkers that work even in the face of concurrency, such as our own work and that proposed by Joshi and Sen [67]. So, for each candidate we recorded whether synchronization primitives (e.g., locks, monitors) were used in the surrounding code. The reasoning here is that any protocol-defining types which use thread synchronization are likely to be used in multi-threaded applications. This may not give us the complete story since classes that are not thread-safe can be used in multi-threaded applications (either because they are used thread-locally or because external synchronization is provided). Still, gathering this information gives us some insight into the ways in which the classes are being used.

### Phase 2: Finding Protocol Usage

In the second phase of the study, we examined how often the protocols we discovered in the first phase were actually used. The input of this phase is the list of protocol methods and protocol types generated in the preceding phase. After running an automated analysis on a suite of code, we were left with a list of all classes that called protocol methods as well as a list of all classes that have fields whose types are protocol types, and an estimate of the number of those classes

that pass their fields' protocols along to their clients. The static analysis itself is rather simple.

**ProtocolUsage** ProtocolUsage is a flow-insensitive static analysis. It proceeds by visiting every method call site in a given code-base. At every method call site, regardless of the receiver, the method binding is statically resolved, and the method's fully qualified name is noted. If the method is in the list of protocol methods, a report is issued, *unless* the method call site is inside the same class as the protocol method being called. Such a call would more accurately be described as an internal interaction rather than a client-provider interaction.

Note that if a class calls protocol methods of its super-class this is considered to be an client interaction with a protocol-defining class, even though at run-time there is only one object. A sub-class can validly be considered to be a client of its super-class, in the sense that a programmer extending another class must be aware of and understand the super-class's rules of use.

ProtocolUsage also looks for instance fields whose types are protocol-defining. In this part of the analysis, at every field declaration, the field's type is resolved. If this type is contained in the list of types defining protocols, a report will be issued.

We are interested in fields of protocol type because they may potentially represent an even closer level of interaction with a protocol-defining type. Since objects referenced by fields are in the heap and may be accessed at any time by member methods, it is more difficult for programmers to obey their protocols than objects that are simply passed and returned amongst methods. Additionally, in our experience it is often the case that classes with fields that define protocols expose those protocols to their own clients.

**Manual Examination** While we did not have the time to investigate all of the fields of protocol type, we did want to get an estimate of the number of classes acting as protocol wrappers, passing on the protocols of their fields to their clients. To this end we took a random sample of the classes containing fields of protocol types (approximately 7%) and we manually investigated those classes to see whether or not the protocols of the fields were passed on to their classes. We recorded whether or not this was the case, and used the rate of protocol passing-on to get a rough estimate for the entire suite of phase two programs.

This is the end of the second phase of our study.

### 2.2.3 Programs Under Analysis

We ran the ProtocolFinder tool on four open-source programs, in order to find out how many protocols they defined. We then ran the ProtocolUsage analysis on those four plus twelve additional programs to determine how often code acts as a client to protocol-defining code.

All of the programs we analyzed in both phases are shown in Table 2.1, along with their sizes and descriptions. With the exception of the standard library and our own analysis framework, Crystal, they all come from the Qualitas Corpus [83]. We attempted to select relatively large, popular open-source programs, and to have a mix of library/framework software as well as end-user applications. Popular programs, we believe, are likely, though not guaranteed, to be well-designed. This will help reduce the risk that the programs we analyzed were poorly designed, and therefore contain abnormally many (or few) protocols. The desire to include both

| Program | Lib./Fwk. or App.? | Version | LOC | Classes (Interfaces) | Description |
|---|---|---|---|---|---|
| **Programs analyzed for protocol definition and usage.** | | | | | |
| Java Standard Library | L/F | jdk1.6.0_14 | 1,012,860 | 8,485 (1,761) | The Java standard library |
| PMD | A | 3.1.1.0 | 26,586 | 396 (27) | A static analysis |
| Azureus | A | 3.3.2 | 102,119 | 900 (354) | A BitTorrent client |
| Eclipse (JDT core) | L/F | 3.3 | 99,691 | 300 (41) | Framework for Java development tools |
| **Additional programs analyzed for protocol usage.** | | | | | |
| ant | A | 1.7.1 | 91,679 | 962 (71) | Tool for building Java programs |
| antlr | A | 2.7.7 | 41,880 | 186 (35) | Lexer/parser generator |
| aoi | A | 2.5.1 | 81,597 | 438 (26) | A 3D modeler |
| columba | A | 1.0 | 68,267 | 982 (109) | A graphical email client |
| crystal | L/F | 3.4.1 | 17,052 | 187 (66) | A framework for writing static analyses |
| drjava | A | 20050814 | 59,114 | 639 (79) | A teaching IDE for Java |
| freecol | A | 0.7.4 | 62,641 | 434 (21) | An open-source clone of Civilization |
| log4j | L/F | 1.2.13 | 13,784 | 178 (16) | A logging library from Apache |
| lucene | L/F | 1.4.3 | 25,472 | 276 (15) | A text search library from Apache |
| poi | L/F | 2.5.1 | 47,804 | 417 (28) | A library for accessing Microsoft documents |
| quartz | L/F | 1.5.2 | 22,171 | 121 (25) | An EJB job scheduling framework |
| xalan | L/F | 2.7.0 | 161,008 | 1,004 (65) | An XSLT XML transformation engine |
| **Total** | 8×A 8×L/F | | 1,933,725 | 15,905 (2,739) | |

L/F=Library or Framework A=Application

Table 2.1: The programs analyzed as part of this study, along with their sizes and descriptions

libraries/frameworks and end-user applications is based on our own intuition. We hypothesized that libraries are more likely to define types with object protocols since they may be wrapping some underlying system resources that is inherently stateful (e.g., sockets and files).

During the course of the study we examined 1.9 million lines of Java, of which 1.2 million was used in the first phase of the study, and of that portion, one million of which is the Java standard library. Examining the Java standard library for object protocols was a high priority. Because of its wide use in most Java programs, knowing which types in the standard library define protocols enables us to analyze client usage of protocols in many more programs. In fact almost all of the client-side protocol usage in our study was usage of standard library types. This makes sense since, for example, Ant is unlikely to use any protocols defined in PMD, Azureus or JDT and we do not know any of the protocols it defines, since it was not part of the first phase of our study.

### 2.2.4 Risks

There are a number of potential risks and threats to validity in the study as designed. Here we discuss some of those risks, as well as some of the mitigating factors.

Some of the most interesting risks in our study are due to our use of static analysis. The use of static analysis is motivated by our desire to examine as large a corpus of programs as possible. Unfortunately, this means the study is subject to the false negative and false positive rates of our static analysis, particularly the ProtocolFinder. For the ProtocolFinder, false negatives are instances where the analysis is run on a piece of code that defines an object protocol and yet the analysis does not report a candidate. False positives are the protocol candidates that are not classified as protocol evidence. False positives are mitigated by manual inspection. Every candidate reported by the ProtocolFinder has been manually inspected to determine whether or not it represents evidence for an object protocol.

However, we can imagine several potential sources of false negatives. The first source is that the ProtocolFinder can only investigate code, and that code must be written in Java. This rules out protocols that are defined by Java interfaces, which contain no code, and native methods, which are written in other languages. We mitigate the former case with our inspection process: when a method is determined to be a protocol method, we note the supertype methods it overrides and add them to our list of protocol methods for use in subsequent phases of the study. For native methods, though, there is not much that we are able to do. Still, out of the 120,085 total methods we analyzed in the first phase of the study, only 739 of them were native methods, suggesting that we might not be missing much.

Another source of false negatives comes from code that does not attempt to detect protocol violations, in other words, protocol-defining code that does not fit the pattern that the ProtocolFinder is looking for. The ProtocolFinder requires code to check or use the value of a receiver field inside a conditional expression and then throw an exception in one branch of the conditional. APIs that fail in an undefined manner when their protocols are violated likely would not fit this pattern.

In order to better explain, let us consider a few examples of protocols that our analysis would not detect. First, our analysis does not detect protocols where something must *eventually* be done. For example of such a protocol, consider the `Connection` class in the `java.sql` package of the Java library. This interface abstracts a database connection and defines a method `close` for closing such connections. Closing a database connection releases the various system resources associated with the connection, and it is generally considered to be best practice to close such connections as soon as they are no longer in used, otherwise resources may eventually be exhausted. Unfortunately, our analysis has no way of detecting such a protocol violation, since there is no exception to be thrown; the implementations of the `Connection` interface never throw exceptions for "connection unused for a long time," since they cannot determine if such an object will be used later in the program.

As another example, consider a class defining an initialization protocol, where violations result in implicit exceptions. The class shown in Figure 2.5 is just such a class. This socket-wrapping class defines an initialization protocol which requires the `setSocket` method to be called before either the `read` or `closed` methods. Critically, a violation of this protocol simply results in a null pointer exception, which will not be detected by our analysis. While we believe

that well-designed code will generally attempt to detect violations of its own protocols, such scenarios are likely a source of real false negatives.

```
1  class SocketWrapper {
2    private Socket s;
3
4    public void setSocket(Socket s) { this.s = s; }
5
6    public byte[] read() {
7      return this.s.read();
8    }
9
10   public void close() {
11     this.s.close();
12   }
13
14   // continues ...
15 }
```

Figure 2.5: This artificial wrapper class defines protocols relating to the initialization of its wrapped object, and to the wrapped object's own protocol.

Similarly, APIs that define protocols due to their delegation to other, protocol-defining APIs may be missed by our ProtocolFinder. The `SocketWrapper` class in Figure 2.5 also illustrates this problem. In this class, because the underlying socket field defines a protocol, the wrapper class also defines a protocol, since calling the `read` method after the `close` method will result in the field throwing an exception. While our analysis does not find these protocols per se, we are attempting to gauge how likely they might be by reporting the number of classes whose fields themselves define protocols. Then, based on a manual examination of a sample of those classes, we estimate the number of unexamined classes that delegate the protocols of their fields.

In our experience, frameworks *do* tend to detect protocol violations and respond with a thrown exception, so we expect our static analysis to succeed in such situations. The problem is that the protocols thrown by frameworks are often far removed from the code that caused the protocol violation in the first place [65]. This means that there is additional burden on us during manual inspection of the results of the ProtocolFinder to ensure that the protocol is correctly understood and associated with the correct types.

Lastly, we have the typical threats of any empirical study: that our selection of programs may be biased, not representative, or too small to draw meaningful conclusions. We have done our best to draw a variety of programs from a respected corpus of popular Java programs [83] that was as large as possible given our time constraints.

| Program | Protocol Candidates | Protocol Evidence | Evidence Classes | Protocol Types | T.S.E.C. | Precision | %E.C. | %P.T. |
|---|---|---|---|---|---|---|---|---|
| JSL | 2,690 | 613 | 195 | 842 | 54 | 22.8% | 2.3% | 8.2% |
| PMD | 32 | 7 | 3 | 10 | 0 | 21.9% | 0.8% | 2.4% |
| Azureus | 136 | 24 | 19 | 32 | 4 | 17.6% | 2.1% | 2.6% |
| JDT | 62 | 4 | 4 | 5 | 0 | 6.5% | 1.3% | 1.5% |
| **Total** | 2,920 | 648 | 221 | 889 | 58 | 22.2% | 2.2% | 7.2% |

T.S.E.C.=Thread-Safe Evidence Classes %E.C.=% Evidence Classes %P.T.=% Protocol Types

Table 2.2: The results of running the ProtocolFinder on the four phase one code bases

## 2.3 Results

In this section we present the results of our study[2], with little additional discussion. Discussion of the results is postponed until Section 2.4. The results of running the ProtocolFinder analysis are discussed in Section 2.3.1, categories of protocols we found are discussed in Section 2.3.2 and the results of running the ProtocolUsage analysis are discussed Section 2.3.3.

The summary is that a little over 2.2% of all classes on which we ran our ProtocolFinder define protocols. 7.2% of all types are considered to define protocols when we include supertypes, and approximately 13.3% of all the classes on which we ran our ProtocolUsage analysis use object protocols as clients. 98% of the protocols we found fit into one of seven simple categories.

### 2.3.1 Protocol Definitions

Table 2.2 contains the results of running the ProtocolFinder analysis on the four code bases used in the first phase of the study. For each code base, the table displays the following information. First, the number of candidates reported by the ProtocolFinder analysis. These varied from around 2,600 for the Java standard library to 32 for PMD. The next column shows how many candidates were manually classified as protocol evidence. The next column shows the number of classes that contained protocol evidence. Next, "Thread-Safe Evidence Classes" displays how many classes containing protocol evidence also use multi-threading primitives. Since we are interested overall in how well our analysis is performing, the next column shows the precision of the ProtocolFinder: the ratio of protocol evidence to protocol candidates. Finally, the last two columns show the percentage of classes containing protocol evidence relative to the total number of classes and the number of protocol types relative to the total number of types. (Recall that our list of protocol types includes classes and interfaces containing methods overridden by methods containing evidence of protocols.) The last row displays cumulative values for each column, along with percentages recalculated from these sums.

[2]All the data gathered during this study can be found at the following location: `http://www.cs.cmu.edu/~nbeckman/research/esopw/`

### 2.3.2 Protocol Categories

Of the 613 candidates that were manually determined to be protocol evidence, we noticed a number of similarities in their structure and intent. In fact, almost all of them could be characterized in one of seven protocol categories, which we will describe in this section. Due to the means by which our analysis produces candidates, the categories we present are largely categories of errors: conditions under which operation of a class will result in an error. Figure 2.6 summarizes the results for each category. One example of each category can be found in Appendix A. More details on each category can be found in Duri Kim's masters thesis [68].

**Initialization (28.1%)** Some types must be initialized after construction time but before the object is meant to be used. In the *initialization* category, calls to an instance method $m$ after construction-time will result in an error unless an initializing method $i$ has been called at least once before. Types may (or may not) allow $i$ to be called multiple times, however, it is a feature of this category that objects cannot become uninitialized after they have already been initialized (i.e., initialization is monotonic).

A typical example of this category is the protocol defined by the Java library class `AlgorithmParameters` in the package `java.security`. After an instance of algorithm parameters is constructed, it is not ready for use until one of its three `init` methods is called. Before initialization, calls to the `toString` method will return null, and calls to `getEncoded` and `getParameterSpec` throw an exception.

**Boundary (7.9%)** Some types force clients to be sure that an instance is still "in bounds." In the *boundary* category, an instance method $m$ can only be called a dynamically-determined number of times. Calling $m$ more times will result in an error. Typically such types will provide some method $c$ to clients so that they can determine if an subsequent call to $m$ is safe, although clients are not required to call it. We can abstract this into a finite number of states by having *is in bounds* and *isn't in bounds* abstract states.

The most widely known example of this category is the iterator. In an iterator, the `next` method can only be called if the iterator is at a location in the iterated collection where there are subsequent items. Iterators provide a method `hasNext` which allow clients to check dynamically if they have reached the end of the collection.

**Deactivation (25.8%)** Some types permit deactivation, after which point instances can no longer be used. In the *deactivation* category, calls to an instance method $m$ will fail after some method $d$ is called on the same instance, and it will always fail for the rest of the object's lifetime. Like *initialization*, types may or may not permit $d$ to be called more than once.

A typical example is the `BufferedInputStream` in the package `java.io`. Once a stream is closed, no further methods can be called on the stream, and it cannot be reopened. A somewhat more interesting example is `FreezableList` from `com.sun.corba.se.impl.ior`. This is a normal mutable list that, at some point during its lifetime, can be made immutable by calling the `makeImmutable` method. After this point mutating methods, like `remove`, can no longer be called. (This is in direct contrast to other immutable lists, like those created by `Collections.unmodifiableList`, which are immutable for the entire object lifetime.)

**Redundant Operation (7.3%)** In the *redundant operation* category, a method $m$ will fail if it is called more than once on a given instance.

For an example of this category, consider the `AbstractProcessor` class, located in the `javax.annotation.processing` package. If the `init` method is called more than once, the second call will fail. One might wonder, given the name of the method, why this is not considered to be part of the initialization category. The answer has to do with the fact that our categories are oriented towards errors. In the initialization category, methods on an object will fail if the object has not already been initialized. Here, the failure occurs when the `init` method is called a second time.

**Dynamic Preparation (8.0%)** Certain methods cannot be called until a different method has been called to ready the object. In the *dynamic preparation* category, an instance method $m$ will fail unless another instance method $p$ is called before it. If we think of types in this category as having two states, *ready* and *not ready*, this category is distinguished from the initialization category in that an object may dynamically change from ready to not ready at numerous points in its lifetime (i.e., it is not monotonic).

The most familiar example of this category is the `remove` method on the `Iterator` interface. An iterator's contract states that the `remove` method cannot be called until `next` has been called, and clients must continue to call the `next` method at least once before each time the `remove` method is called.

**Type Qualifier (16.4%)** Some types disable certain methods for the lifetime of the object. In the *type qualifier* category, an object instance will enter an abstract state $S$ at construction-time which it will never leave. Calls to an instance method $m$, if it is disabled in state $S$ will always fail. This category is so-named since it is similar in spirit to flow-insensitive type-qualifiers [45] (or alternatively, type refinements [38]).

Protocols in this category show two distinct behaviors. In some cases, the abstract state that newly constructed instances inhabit can be set by parameters to the constructor. For example, instances of the `ByteBuffer` type in the `java.nio` package may or may not be backed by a byte array. Whether or not they are depends solely on whether or not a backing array was provided at construction-time. If one was not provided, any calls to the `array` method will fail with a run-time exception. In other cases, the instantiating class itself determines the abstract state that all instances will inhabit, relative to the abstract states defined in a super-type. For example, consider the instances returned from calls to `Collections.unmodifiableList` in the Java standard library. All such instances are *unmodifiable* relative to the super-type `List`, which permits both mutable and immutable lists. In both case, clients must be aware of which methods are enabled.

**Domain Mode (4.8%)** The *domain mode* category captures protocols in which certain methods can be called only when the object is in a particular domain-specific mode. Typically there is a way to set the mode of an object through a method like `setMode(mode)` or a set of methods like `setModeToX()`. Unlike the type qualifier category, modes can be changed dynamically. The

preparation category is sometimes similar in structure, but the domain mode category differs in that there is an explicit intent to represent multiple modes of the object.

As an example, consider the `ImageWriteParam` class in the `javax.imageio` package. An image may be written with or without compression. The ImageWriteParam object can be used to control whether and how compression is used. It has a compression mode, which may be no compression, explicit, or writer-selected. In the explicit mode, the ImageWriteParam object controls the compression type, and `setCompressionType` may be called only in this mode. In other modes, this method will throw an exception, because either no compression is to be used at all, or the writer is intended to select the compression type.

**Others (1.9%)** Finally, there were a smattering of protocols that did not fit any of the previously-mentioned categories, although even these protocols themselves have certain similar characteristics. As examples, we encountered a few instances of types that defined methods that must be called in strict alternation (a single call to method $A$ enables a single call to method $B$ and vice versa). We also found a limited number of protocols that we would describe as *lifecycle* methods, where a type defines more multiple abstract states through which an object transitions monotonically during its lifetime. For example, the `GIFImageWriter` and `JPEGImageWriter` classes in the Java imageio library seem to have this behavior. While we did not encounter many lifecycle protocols, our own experience with Object-Oriented frameworks suggests that they may be more common elsewhere.

| Category | Protocol Evidence | % |
|---|---|---|
| Initialization | 182 | 28.1% |
| Deactivation | 167 | 25.8% |
| Type Qualifier | 106 | 16.4% |
| Dynamic Preparation | 52 | 8.0% |
| Boundary | 51 | 7.9% |
| Redundant Operation | 47 | 7.3% |
| Domain Mode | 31 | 4.8% |
| Others | 12 | 1.9% |

Figure 2.6: Categorization of each of the 648 reports issued by the ProtocolFinder that were evidence for actual protocols.

### 2.3.3 Protocol Usage

Table 2.3 shows the results of running the ProtocolUsage analysis on the sixteen candidate programs from phase two of the study. The goal here is to see how often classes act as clients of other protocol-defining types. The table contains the following information: The first column after the list of programs is the number of classes in that program that contain calls to protocol methods. The next column shows the percentage of classes in each program that use protocol methods. These numbers range from 4% of all classes using protocols, on the low end, to 28% of all classes on the high end. The next two columns show the number and percentage of classes

| Program | Classes Calling Protocol Methods | % | Classes w/ Prot. Fields | % | Exposes Protocol Rate | Est. Classes From Total |
|---|---|---|---|---|---|---|
| JSL | 1012 | 12% | 1082 | 13% | 15% | 157 |
| PMD | 85 | 22% | 29 | 7% | 0% | 0 |
| Azureus | 198 | 22% | 763 | 8% | 31% | 234 |
| JDT | 13 | 4% | 18 | 6% | 0% | 0 |
| ant | 269 | 28% | 187 | 19% | 20% | 37 |
| antlr | 20 | 11% | 16 | 9% | 0% | 0 |
| aoi | 25 | 6% | 37 | 8% | 0% | 0 |
| columba | 120 | 12% | 246 | 25% | 8% | 18 |
| crystal | 9 | 5% | 2 | 1% | 0% | 0 |
| drjava | 49 | 8% | 107 | 17% | 0% | 0 |
| freecol | 94 | 22% | 117 | 27% | 0% | 0 |
| log4j | 39 | 22% | 32 | 18% | 0% | 0 |
| lucene | 30 | 11% | 27 | 10% | 0% | 0 |
| poi | 41 | 10% | 13 | 3% | 100% | 13 |
| quartz | 16 | 13% | 10 | 8% | 0% | 0 |
| xalan | 91 | 9% | 142 | 14% | 13% | 17 |
| **Total** | 2111 | 13% | 2141 | 13% | 17% | 356 |
| **Excluding JSL** | 1099 | 15% | 1059 | 14% | 18% | 196 |

Table 2.3: The results of running the ProtocolUsage analysis on the sixteen candidate code bases.

that have fields whose types are protocol-defining types. The column, "Exposes Protocol Rate" shows the percentage of the classes with protocol fields that were found to expose the protocols of those fields to their own clients, of the 7% of classes with protocol fields that we sampled. The column, "Est. Classes From Total" is an estimate of the total number of classes that expose protocols defined by their fields based on this rate. The last two rows show the totals and cumulative percentages for the entire suite, as well as the numbers excluding the Java standard library.

We were also interested in finding out which protocol methods were being called most frequently, and Figure 2.7 summarizes this information. This table contains a list of the twenty most frequently-called protocol methods. During our examination of the sixteen open-source code bases used in phase two, we found 7,645 calls to protocol methods. We took all the protocol methods that were called, and ordered them by how many times they were called. Figure 2.7 shows the twenty most frequently called protocol methods along with the number of times that method was called in our candidate programs and the percentage of the 7,645 protocol method calls that particular method constitutes. For example, the `next` method of the `Iterator` interface was the most-frequently called protocol method in our study. Of the 7,645 calls to protocols we found, over 2,200 were calls to `Iterator.next`, almost 30% of the calls.

| Method | Calls | % Calls |
|---|---|---|
| java.util.Iterator.next() | 2226 | 29.11% |
| java.util.Enumeration.nextElement() | 1022 | 13.37% |
| java.lang.Throwable.initCause(Throwable) | 850 | 11.12% |
| org.w3c.dom.Element.setAttribute(String,String) | 460 | 6.02% |
| java.util.Iterator.remove() | 211 | 2.76% |
| java.io.Writer.write(int) | 182 | 2.38% |
| java.io.OutputStream.write(int) | 165 | 2.16% |
| java.io.InputStream.read() | 162 | 2.12% |
| sun.reflect.ClassFileAssembler.cpi() | 138 | 1.81% |
| org.omg.CORBA.portable.ObjectImpl._get_delegate() | 90 | 1.18% |
| java.io.InputStream.read(byte[],int,int) | 89 | 1.16% |
| java.util.ListIterator.next() | 80 | 1.05% |
| java.io.Writer.write(char[],int,int) | 77 | 1.01% |
| java.io.PrintWriter.flush() | 76 | 0.99% |
| java.io.OutputStream.flush() | 75 | 0.98% |
| java.nio.Buffer.checkIndex(int) | 65 | 0.85% |
| javax.swing.text.AbstractDocument.readUnlock() | 61 | 0.80% |
| org.w3c.dom.Element.setAttributeNS(String,String,String) | 59 | 0.77% |
| java.io.OutputStream.write(byte[],int,int) | 57 | 0.75% |
| java.io.InputStream.reset() | 45 | 0.59% |

Figure 2.7: The 20 most-frequently called protocol methods, out of a total of 7,645 calls to protocol methods, and percentage occurrence of each method relative to the total.

## 2.4 Discussion

After running our experiment, we noticed some interesting results. Protocols were defined with small, but significant frequency, and almost all of those protocols fit within a small number of categories. All of the protocols we expected to find we did find, which gives us some confidence in our approach. And a significant number of classes in our study use protocols as clients, even though almost all of the protocols we were looking for were defined in the Java standard library. Interestingly, but not surprisingly, there are a few protocols that are much more widely used than others.

### 2.4.1 Sanity Check

As discussed in Section 2.2.4, we curious about the ProtocolFinder's false-negatives: protocols that were defined in the code under analysis but not discovered due to the design of the analysis. One quick sanity check we can do is to make sure that all the protocols we already know about are found by our analysis. This is not perfect, since our ProtocolFinder was designed with these protocols in mind. Still, it is somewhat comforting to see that all of the protocols we have encountered in our previous work, and in similar related work are found by our analysis.

We expected to see sockets, files, streams and iterators in our results, since those types are widely discussed in related work. And with the exception of the actual `java.io.File` class,

which does *not* define a protocol, we were not disappointed. Socket, Readers, Writer, Streams and all their related classes did turn up in our analysis. (Interestingly, `ZipFile` does define an Open/Closed protocol.) We were also aware of the `Throwable` and `Timer` protocols from our own work.

Additionally, we were happy to see that well-known protocol-defining interfaces, like `Iterator`, were discovered through our process, since, for interfaces, the ProtocolFinder has no code to examine.

### 2.4.2 Widely Used Protocols

We were quite interested, although not surprised, by the twenty most frequently called protocol methods, shown in Figure 2.7. The iterator protocol, a protocol examined in several recent works [15, 76], appears at the top of the list, and the `next` method of the iterator protocol accounts for nearly a third of all protocol method calls.[3]

While this seems rather uninteresting, it does suggest two points. One, that the time spent evaluating protocol checkers against the iterator interface may be well-spent, since a good iterator-checker can check a large portion of the protocols that are used in practice. Second, all of the calls recorded are actual calls to `Iterator.next`, and not instances of Java 5's enhanced for loop. While at present, these do represent actual protocol uses, where the client needed to understand the Iterator's protocol in order to use it, one suspects that many of these calls could be replaced by the enhanced for loop, which would dramatically reduce the number of protocol clients we observed. (The same cannot be said for calls to `Iterator.remove`.)

The remaining frequently called methods quickly drop off in the frequency of their use. The most-frequently called list leaves something like forty percent of all protocol method calls off. This suggests that most protocols, like most APIs in general, have a small number of clients. Most of the commonly used protocols are quite recognizable: readers, writers, streams and certain collections defining abstract states. Interestingly, when we remove recognizable types (e.g., streams, sockets, files, iterators, throwables and their subclasses) we found that what was left accounted for 21% of all protocol usage. This means there is still a fair amount of use of non-obvious protocols.

### 2.4.3 Protocol Categories

We were pleasantly surprised to discover that a small number of categories (seven) could be used to classify almost all of the protocols that we encountered (98%). This is useful because it suggests a new evaluation criteria for developers of typestate checkers. If a typestate checker can verify protocols from each of the seven categories, it suggests that checker will likely work on most of the protocols it is every likely to encounter. Of course, things are likely not so simple. For many typestate analyses, it is the context in which a protocol is used rather than the complexity of the protocol that makes a piece of code easier or more difficult to verify. For example, experience

[3]It is worth noting that the `hasNext` method, which we would generally consider to be part of the Iterator's protocol, does not show up at all in our list of protocol methods. This is due to the fact that the implementations of `hasNext` do not normally partake in protocol violation detection by throwing an exception.

verifying protocols (Bierhoff et al. [16] and Chapter 6 of this thesis) has shown that whether or not an object is aliased has a lot to do with the difficulty of protocol verification.

It is also interesting that the categories produced during this study have the flavor of "protocol primitives," and this may have something to do with how the study was carried out. To illustrate, one may have noticed that none of the categories that we found have more than two abstract states. Yet this does not mean that none of the types we investigated had more than two abstract states. Our study proceeded by investigating each location of interest as determined by the ProtocolFinder. We tried to understand only enough of the implementation to determine whether or not we were seeing evidence for a protocol, the state that the class should be in in order not to have that particular exception thrown, and which state the class is in if the exception is thrown. But classes can have different pieces of a protocol that fit into different categories or even multiple protocol pieces that are all in the same category. As an example of the latter case, consider the `Socket` class in `java.net`. A socket instance can be open or closed, its "write-half" can be open or shut down. Both aspects of the protocol are categorized as *deactivation check* protocols, but if one to is to consider the class' protocol in total, it would have at least four abstract states.

All of this is to say that there may be interesting characteristics shared by protocol-defining types that are not captured by our categories. Coming to a better understanding of protocols at a larger level of granularity, while an interesting topic for future work, is out of the scope of this study.

### 2.4.4 Other Observations

A number of other points can be made by examining the results of our study.

One point suggested by the data is that protocol use (13% of all classes) is more common than protocol definition (7.2% of all types). This information is hardly surprising. Notice, however, that the percentage of classes that use protocols is not *that* much greater than the percentage of types that define them. This could be due to a design principle of localizing state in an application. Or perhaps it just means that there are a lot of types in the Java standard library that are not commonly used.

Still, the data suggest that client-side protocol checking may be more important than implementation-side checking. Certain protocol-checking approaches have the ability to verify both the correct use of protocols by clients and the correct implementation of protocols by their providers. Such is the case for the approach presented in this thesis and that of Bierhoff and Aldrich [15]. While provider-side checking may be important in some situations, a good client-side protocol checker may give programmers the most bang for the buck.

In Table 2.3 we showed that 13% of all classes have fields whose types are protocol types. From the 7% of those classes we manually examined in our random sample, 17% of them were found to expose the protocols of their fields to their clients. Extending this rate to the entire set of classes with protocol fields, we estimate that something like 356 of the classes in the phase two programs define object protocols simply because of the ways in which their fields must be used. This represents about 2% of all of the classes we examined in the entire study, and could represent an additional, significant increase in the percentage of protocol-defining types.

Of all the classes defining protocols, the percentage implemented with synchronization primitives was significant. Out of 221 classes containing protocol evidence, 58 of them, or 26.2%

were designed to be accessed by multiple threads concurrently. If protocol checking is considered an area of research interest, this suggests that those checkers should be designed with multi-threading in mind. This evidence supports the rest of the work done in this thesis.

We did not observe conclusively that protocols were more likely to be defined by libraries and frameworks than by applications. However, the Java standard library when considered separately, has a much higher percentage of its types classified as protocol-defining (8% vs. approximately 2%). There could be some truth to the idea that code wrapping underlying system resources is more likely to define protocols. However, given our process of gathering protocol types, it might alternatively suggest that the standard library has a deeper type hierarchy.

For protocol usage, there was some difference observed. In programs that we classified as applications, 17.4% of classes acted as clients of protocol-defining methods. For library and framework code, that rate was 11.4%.

In general, we were pleasantly surprised by the variety of types that define protocols. As evidenced by the small number of protocol categories, these protocols were often quite similar. Still, we found protocols in a wide variety of types. To name just a few, we found protocol defining types in the following areas:

**Security** `com.sun.org.apache.xml.internal.security.signature.Manifest`, `java.security.KeyStore`

**Graphics** `java.awt.Component.FlipBufferStrategy`, `java.awt.dnd.DropTargetContext`

**Networking** `javax.sql.rowset.BaseRowSet`, `javax.management.remote.rmi.RMIConnector`

**Configuration** `javax.imageio.ImageWriteParam`, `java.security.AlgorithmParameters`

**System** `sun.reflect.ClassFileAssembler`, `java.lang.ThreadGroup`

**Data Structures** `com.sun.corba.se.impl.ior.FreezableList`, `java.util.Vector`

**Parsing** `net.sourceforge.pmd.ast.JavaParser`, `org.eclipse.jdt.internal.compiler.parser.Scanner`

This list is not exhaustive, by any means. This at least can help answer one of the questions that helped to motivate this study: Are there *any* protocol types beyond files, sockets and iterators? We can say, confidently, that the answer is yes.

## 2.5 Related Work

The problem of finding classes that define protocols is one of protocol inference, and there has been some work in this area. The two most closely-related studies were done by Whaley et al. [97] and Weimer and Necula [96].

Both Whaley et al. [97] and Alur et al. [5] have developed effective tools for statically inferring protocol definition. Whaley et al. [97] present a dynamic and a static analysis for inferring object protocols. Their static analysis is inspired by the same reasoning that ours is, and the description contains an in-depth discussion of the practice of "defensive programming," which

is what we have described here as detection of protocol violations. The dynamic analysis they propose can infer more complex protocols than the static analysis. While our experiments cover some of the same ground as theirs (both examine the Java standard library) our focus is different. Their primary focus is on the analyses themselves, with the frequency and character of the protocols taking a back-seat. Their largest studies were performed using the dynamic analysis, and so in some ways are not comparable since not all of lines of code are executed during dynamic analysis. Our best estimate is that their study covered approximately 550 thousand lines of source, compared with 1.2 million lines of source covered in phase one of our study. Numbers are only reported for the Java standard library experiment. They report that 81 of 914 classes define protocols. Our experiments for version 1.6.0_14 report that 195 of 8,485 classes define protocols, and show how much the Java standard library has grown since version 1.3.1! Still, their work contains some discussion of the relevant methods and interesting features of these object protocols. Our work contains a more systematic description of the protocols encountered, including a classification of those protocols. Lastly their static analysis seems to be more precise. It can detect protocol violations that result in null pointer exceptions, which ours cannot.

Alur et al. [5] propose a related static protocol detector that also seems to be more precise than ours. They also looked at the Java standard library, albeit just a handful of classes. While either of these static analyses might have made a better candidate for our own study, neither are publicly available.

Weimer and Necula [96] performed a study on open-source software that in some ways is similar to ours. In their work, they were looking for violations of resource-disposal protocols. For example, a connection to a database that *must* be closed eventually, ideally as soon as it is no longer needed. They examined over four million lines of open-source Java code and found numerous violations of these sorts of protocols. This study, while quite interesting, differs from ours in a number of ways. First off, their focus was on finding violations of protocols rather than characterizing the nature and use of protocols (correct or otherwise) as we have done. While they did look for protocol violations, they made no systematic attempt to discover automatically the types that define such protocols. Rather, they started their experiments with a known list. Additionally, their notion of protocol and our notion of protocol do not quite overlap. They consider protocols to be instances on which some operation must eventually be performed. While most would consider this to be a protocol, it does not fit into the definition we presented in Section 2.2.1. The protocols we consider, protocols in which calling a method at the wrong time will lead to an error, are not considered in their work.

Recent work has been done in the area of dynamic API protocol inference [60, 101]. It is our position that dynamic inference is inappropriate for our needs, since using these analyses requires, at a minimum, test cases to exercise parts of code that use protocols. In our attempt to find as many protocols as possible in as much code as possible, finding test cases has proved to be quite difficult. That being said, one advantage of these approaches is that they do not require specific protocol patterns (e.g., code that throws an exception) in order to detect a protocol. Rather, the work by observing method call orderings that frequently occur. This allows protocols to be inferred that our ProtocolFinder often cannot.

Other researchers have developed an approach that can automatically infer temporal specifications by statically analyzing source code [95]. In this approach, models of correct object behavior are extracted from source code, and from these models, CTL specifications are auto-

matically generated. These specifications in turn are used to automatically find violations. Such as tool could be used as an alternative for our own ProtocolFinder.

## 2.6 Future Work

Our study suggests a number of potential avenues for future work. For one, the simple static analysis, ProtocolFinder, developed for this study, while useful, is not sound with respect to our own definition of object protocol. Better analyses will likely find even more protocol definitions in the same code base. Alternatively, widening the definition of object protocol to include more object behaviors, will also likely result in finding more object protocols in the same code base, and a wider definition may be of interest to certain researchers and practitioners.

As discussed in Section 2.4.3, our current protocol categories are in some sense "micro-categories:" primitive categories from which larger behavioral patterns might emerge. An interesting task for future work is to examine these larger behavioral entities to see if they share common characteristics.

Finally, even if object protocols are common, an interesting question to ask is whether or not they lead to program defects. Studying the correlation between protocol definition and use in a code base and the quality of that code may help to answer this question.

## 2.7 Conclusion

In this chapter we presented an empirical study that examined several popular open-source Java programs. The goal was to determine the true nature of object protocols; how often they are defined, how often they are used, and in what way those protocols are similar. In order to examine as much code as possible, which can help us draw broad conclusions, we developed two static analyses, ProtocolFinder and ProtocolUsage, which help us find where protocols may be defined and where they are used. ProtocolFinder in particular may be subject to false negatives, but regardless was able to find many of the most commonly discussed object protocols.

We found that object protocols are occasionally defined (on average, 7.2% of all types were found to define protocols) but more commonly used (on average, 13% of classes acted as clients of protocols). A small number (seven) of rather simple protocol categories were used to classify almost all of the protocols we found.

# Chapter 3

# Approach: A Type System for Correct Concurrent API Usage

*Keep it simple. The more you say, the less people remember.*

---

This chapter presents our approach for statically checking the concurrent usage and implementation of object protocols. Our checking methodology is formalized as a type system. It extends the basic access permission methodology proposed by Bierhoff and Aldrich [15] so that they are handled soundly in the face of concurrent access. A previous version of this chapter appeared at the OOPSLA 2008 conference [10]. Chapter 4 contains a proof of type safety for the language described in this chapter.

## 3.1 Introduction

For the scope of this work, we consider how race conditions can lead to misuse of object protocols. Our goal is to statically prevent races on the abstract state of an object, as well as violations of an object's concrete state invariants due to concurrent access. Throughout this chapter we will use a concurrent queue as a running example. A client-side use of this queue is shown in Figure 3.1. Part of its implementation is shown in Figure 3.2. The queue itself comes from the Axl-Lucene[1] open-source application. The client was written for the purposes of explaining our analysis.

There are two ways in which protocols of thread-shared objects can be abused that we want our static analysis to catch. First, as clients use objects that define protocols, they may inadvertently create race conditions on the abstract state of those objects. Second, in the implementation of those protocols, the objects themselves may transition non-atomically from one state to another, which may cause other threads to see fields of that object in an inconsistent state. In an attempt to illustrate the first point, Figure 3.1 shows a client of the `Blocking_queue` class using a queue to share information between two threads. This class is designed to be used by a single

[1] `http://packages.ubuntu.com/dapper/web/axyl-lucene`

producer thread that inserts items into the queue, and multiple consumer threads which remove those items from the queue. In order to prevent the consumer threads from waiting indefinitely for items that will never be inserted, the queue defines a simple protocol. When the queue is *closed* by the producer, this is the signal that no further items will be inserted.

Unfortunately, and even though the implementation `Blocking_queue` uses correct synchronization, there is a race condition on the abstract state of the queue. In between the consumer thread's call to `is_closed` and its call to `dequeue`, it is possible for the producer to close the queue, causing the consumer's call to `dequeue` to throw a run-time exception. These issues are sometimes referred to as "check-then-act" errors or violations of atomicity. The author of this class alludes to these problems in the class' documentation, saying the `is_closed` method, "is inherently unreliable in a multithreaded situation" and that to achieve correct behavior the client, "must synchronize on the queue." While the comments are helpful, because the protocol is a very real part of the object's interface it would be nice to ensure at compile-time it is used correctly.

```
1  final Blocking_queue queue = new Blocking_queue();
2
3  (new Thread() {
4    @Override
5    public void run() {
6      while( !queue.is_closed() )
7        System.out.println("Got object: " + queue.dequeue());
8    }}).start();
9
10 for( int i=0;i<5;i++ )
11   queue.enqueue("Object " + i);
12
13 queue.close();
```

Figure 3.1: In this client-side use of a concurrent queue, there is a race condition on the open/closed state of the queue between lines 6 and 7.

It is also important to verify that state transitions for thread-shared objects are performed atomically. Figure 3.2 shows an implementation of the `close` method of the `Blocking_queue` class that does not atomically transition from the current state to the closed state. (Note that this is *not* the actual implementation used in the Axyl-Lucene project, but is used for illustrative purposes.) A design invariant of the queue is that when it is closed, the `elements` field, which holds a list, is to be set to null, and the `closed` field must be set to true. If the queue is thread-shared, it must transition to the closed state atomically, otherwise there is the risk of a null pointer dereference. The implementation of the `is_closed` method only checks the `closed` field, and the `dequeue` method dereferences `elements` without checking whether or not it is null. Therefore, two threads racing on the queue, one to close and one to dequeue, could inadvertently cause a null pointer dereference even if the consumer is properly accessing the queue.

In this chapter, we describe a Java-like programming language whose type system statically prevents misuse and incorrect implementation of object protocols in concurrent systems. Up to the invariants that are specified by the programmer, this type system prevents race conditions and guarantees that invariants are reestablished at the end of method bodies, even in the face

```
1  synchronized boolean is_closed() {
2    return closed;
3  }
4
5  synchronized Object dequeue() {
6    if( elements.size() > 0 ) {
7      ...
8    }
9  }
10
11 void close() {
12   synchronized(this) { elements = null; }
13   synchronized(this) { closed = true; }
14 }
```

Figure 3.2: In this implementation of the queue class, there is a bug in the `close` method because the queue does not transition atomically from the open to the closed state.

of concurrent access to an object and its fields. Our system uses typestate [90] specifications as the language of invariants, and object permissions [25] to approximate whether or not an object can be thread-shared. Our work builds upon recent work for verifying typestate of aliased objects [15].

The contributions of this language are as follows:

- We have developed a programming language that can soundly check protocol usage in concurrent programs. The type system of this language guarantees that there are no race conditions on the abstract state of an object. If a method call requires the receiver object to be in some state, at run-time the object will be in that state. Furthermore, the specified invariants of these abstract states will be preserved, even in the face of concurrent access.
- In our approach, we reinterpret access permissions, which were previously used as an alias-control mechanism, as an approximation of the thread-sharedness of a location in memory. Our solution is an improvement over existing, lock-based approaches [64, 84] because it does not impose hierarchical restrictions on aliasing, and because our specifications are more compositional.
- We have proved soundness for a core subset of this language in Chapter 4.
- We have developed Sync-or-Swim, a static analysis for Java programs based on the type system of this language.

Existing work on data race detection [24, 39, 81] does a good job of ensuring that access to thread-shared memory is protected by locks or other mutual exclusion primitives, but it does not prevent a program's threads from interleaving in ways that destroy application invariants.

Preventing thread interleavings that destroy program invariants is an important goal, because invariants allow programmers to reason about the behavior of their programs. Toward this goal, several earlier works [64, 66, 78, 93] attempt to statically prevent or prove impossible thread interactions that might invalidate invariants. Compared to these approaches, our work allows for a larger variety of thread-sharing patterns, and additionally helps to ensure the proper use of

typestate, an abstraction of object state that forms an implicit but unchecked interface in many object-oriented programs.

The rest of the chapter proceeds as follows. In Section 3.2 we describe our technique at an informal level, using our concurrent queue as a running example. By the end of this section, readers should understand the intuition behind our approach. Section 3.3 describes the formal language in greater detail. Section 3.4 describes Sync-or-Swim, the implementation of the approach. In Section 3.5 we discuss the wealth of existing work in verification of concurrent software. In Section 3.6 we discuss how we would like to improve our technique, and in Section 3.7 we conclude.

## 3.2 Overview

At a high level, our approach is as follows:

- We use typestate specifications on methods and classes to say which abstract state an object must be in before calling a method on it, and which concrete states an object's fields must be in at the end of a method call.
- Object references are annotated with access permissions which describe how an object pointed to by a reference is shared. Permissions were originally proposed as a means for guaranteeing the non-interference of threads. More recently permissions have been used to control aliasing. Now we reinterpret the same interfering permissions to describe how threads share objects.
- Finally, we track the state of objects as they flow through method bodies, discarding knowledge about the state of an object when the reference to that object indicates it may be modified by other threads *and* we cannot determine statically that its lock is held.

In the next several sub-sections, we describe each part of the process in greater detail.

### 3.2.1 Object Protocol Modeling

Our approach uses typestate [90] as the language of behavioral specification. A specification tells the system which application-specific logic must be upheld in the face of concurrent access.

Typestate specifications allow programmers to develop abstract protocols describing a method or class' behavior. The abstractions take the form of state-machines, an abstraction with which most programmers are familiar. Figure 3.3 is a model of the protocol defined by the class `Blocking_queue`. Here is how we can interpret this diagram: When the queue is constructed, it starts out in the "OPEN" state. Both this state and the "CLOSING" state are sub-states of the "STILLOPEN" state. Whenever the queue is in the "STILLOPEN" state, objects can be dequeued by consumer threads, but they can only be enqueued by the producer when the queue is in the "OPEN" state. If the queue is open the consumer can call the `close` method to close it, transitioning it to the "CLOSED" state. Alternatively, it can enqueue a final item, and the queue will stay open until a consumer dequeues that item. At any time, threads can call the `is_closed` method to determine, from its return value, whether or not the queue is closed.

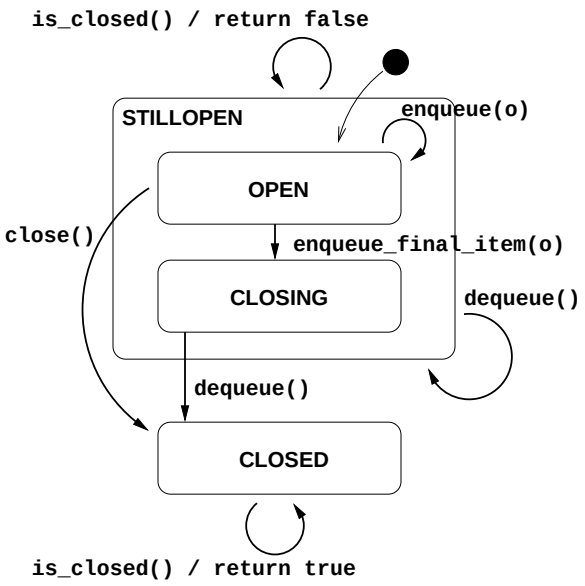


Figure 3.3: A simplified depiction of the protocol defined by the `Blocking_queue` class.

Existing work has been done in statically verifying that an object's behavior will conform to its typestate specification at run-time [31]. Our work, in particular, adapts the approach of Bierhoff and Aldrich [15] for use in concurrent settings. In the approach proposed by Bierhoff and Aldrich, object states are tracked statically using linear logic predicates [48] which treat object state information as a resource that can be consumed and transformed. Methods that transform the state of an object will consume its old state, and return a new state, and the type of the reference to that object will reflect its new state in subsequent lines of code.

State names are typically defined and given meaning by the programmer. However, this paper will mention one privileged abstract state, alive, which is always implicitly the root of an object's state hierarchy.

### 3.2.2 Access Permissions

Access permissions [15] are a means of associating object references with (a) the state of the object referenced and (b) a succinct summary of the ways in which that object can be aliased. Without some information about the aliasing behavior of the program a sound analysis would have to be extremely conservative, rendering it difficult to use. In this section we will show how access permissions can approximate information on whether or not an object is thread-shared, and why this is a sound approximation.

**Permission Kinds** The access permissions system that we use has five different permission types, each one describing whether or not the object is aliased, whether the given reference can be used to modify the object, and whether other references to the object, if they exist, are allowed to modify the object. These permissions are named as follows:

- unique permission to an object indicates that this reference is the sole reference to an object in the program. This is the same as a linear reference in other type systems [94].
- full permissions are exclusive read/write references that can coexist with any number of read-only references.
- immutable permissions are associated with references that point to immutable objects. Any number of these references can point to the same object, but no reference may have modifying access.
- pure permissions are read-only permissions to objects that may be modified through other references.
- share permissions are associated with references that can read and write objects that can also be read and modified by any number of other references in the system. This is the least restrictive permission, and is effectively the default in languages like Java.

A bit later we will show how permissions of one kind can be soundly split into permissions of another kind. This is the controlled mechanism by which new aliases will be introduced into a program while still preserving the meaning of each of the five permission kinds.

**Hierarchies, Dimensions and Guarantees** We inherit from Bierhoff and Aldrich [14] a powerful system of state hierarchies and dimensions, based loosely upon Harel statecharts [52]. State hierarchies allow states to be refined into a number of mutually exclusive sub-states. These sub-states can provide a more refined description of the current state of an object, much in the same way a sub-type may provide a more refined notion of the operators available on an object. For example, in Figure 3.3 we showed how the "STILLOPEN" state of the concurrent queue could be further refined to a "OPEN" state and a "CLOSING" state.

State dimensions allow a state to be divided into sub-parts, each defining its own states. The interesting feature is that the object can inhabit one state in each dimension at the same time. This is useful for modeling objects that define multiple state machines, each of which acts independently.

*Weaker* permissions, like share and pure can be made more powerful through the use of state guarantees. A state guarantee is a sort of contract guaranteeing that an object will never leave a particular state, even if it may transition amongst substates of that state.

Hierarchies, dimensions and guarantees all manifest themselves in our permission syntax:

$$\mathsf{unique}(\mathtt{q}, \mathsf{STILLOPEN})\ \mathtt{in}\ \mathsf{OPEN}$$

This permission tells us that a variable q in a program points to an object with unique permission, meaning that no other references have permission to access this object. Furthermore, the permission is guaranteed in the STILLOPEN state (although this does not matter much since there are no other permissions to depend on the guarantee) and the object is currently in the OPEN state. In Section 3.3 we will further enhance our permissions with fractional values.

**Permission Splitting** The access permissions are arranged in a partial order and can be *split* in order to create other permissions to the same object. This is necessary because when an object constructor is called, a single unique reference is returned, but we may want to then create multiple references to distribute to different parts of the program.

The six splitting rules, defined as follows, say that the access permission on the left-hand side can be soundly traded for the access permission(s) on the right-hand side, thus increasing the number of aliases to an object in the program:

$$
\begin{array}{rcl}
\mathsf{unique}(r,g)\ \mathtt{in}\ A & \Rrightarrow & \mathsf{full}(r,g)\ \mathtt{in}\ A \\
\mathsf{unique}(r,g)\ \mathtt{in}\ A & \Rrightarrow & \mathsf{immutable}(r,g)\ \mathtt{in}\ A \\
\mathsf{unique}(r,g)\ \mathtt{in}\ A & \Rrightarrow & \mathsf{share}(r,g)\ \mathtt{in}\ A \\
k(r,g)\ \mathtt{in}\ A & \Rrightarrow & k(r,g)\ \mathtt{in}\ A \otimes k(r,g)\ \mathtt{in}\ A \\
 & & \text{where } k \text{ is } \mathsf{immutable}, \mathsf{share} \\
k(r,g)\ \mathtt{in}\ A & \Rrightarrow & k(r,g)\ \mathtt{in}\ A \otimes \mathsf{pure}(r,g)\ \mathtt{in}\ A \\
 & & \text{where } k \text{ is } \mathsf{full}, \mathsf{immutable}, \mathsf{share}, \mathsf{pure} \\
k(r,g)\ \mathtt{in}\ A & \Rrightarrow & k(r,A)\ \mathtt{in}\ A \\
 & & \text{where } k \text{ is } \mathsf{unique}, \mathsf{full}
\end{array}
$$

Here $g$ is the state guarantee and $A$ is the current abstract state of the object referenced by $r$. The $\otimes$ symbol is called the "tensor," and is used to combine multiple permissions.

These six splitting rules are the only splitting rules that are allowed, because they are the only (useful) rules that preserve the meaning of each of the five permission kinds. Consider what would happen if an additional rule were added, one which would allow a unique permission to be split into two unique permissions. The result would be two different permissions to the same object, each of which guarantees that it is the only permission to the object. This is a contradiction, and therefore the splitting rule is not allowed.

In the formal language, it is the responsibility of the linear logic proof judgment to automatically determine when and how permissions should be split into other permissions. If several expressions in a method require different permissions to the same reference, the implementation of this judgment must solve these constraints by splitting the permission in an appropriate way. In our implementation, this is performed by a constraint solver [13, Ch. 5].

While it is not needed for the verification of our queue example, permissions can also be reassembled thanks to fractional values, discussed in Section 3.3.

### Method Specifications

Now that we have seen access permissions, we can string them together with linear logic connectives to create specifications. The $\longrightarrow$ connective is called "linear implication," and is used to specify method pre- and post-conditions. Predicates on the left-hand side form the method pre-condition, and those on the right-hand side form the post-condition. Predicates in the pre-condition are consumed and cannot be reused unless explicitly returned by the post-condition. Linear conjunction ($\otimes$) is used when we wish to say that multiple objects must be in specific states at the same time, and additive disjunction ($\oplus$) is used when one of several state predicates may be true. Additive conjunction ( $\&$ ) defines an either/or choice that the client of a specification gets to decide.

We have annotated the methods of the `Blocking_queue` class with behavioral annotations in Figure 3.4. Each specification tells the client the requirements on the abstract state of the receiver before the call and what the resulting state will be when the call has returned. For example, in order to call the `enqueue` method, a client must have a full permission to the queue in question

```
class Blocking_queue {
  Blocking_queue() : 1 ⟶ unique(this) in OPEN { ... }

  void enqueue(Object o) : full(this) in OPEN ⊗ share(o) ⟶ full(this) in OPEN
  { ... }

  void enqueue_last_item(Object o) : full(this) in OPEN ⊗ share(o) ⟶
                                     pure(this) in CLOSING
  { ... }

  Object dequeue() : pure(this) in STILLOPEN ⟶ share(result) ⊗ pure(this)
  { ... }

  boolean is_closed() : pure(this) ⟶ (result = true ⊗ pure(this) in CLOSED) &
                        (result = false ⊗ pure(this) in STILLOPEN)
  { ... }

  void close() : full(this) in OPEN ⟶ full(this) in CLOSED
  { ... }
}
```

Figure 3.4: Specifications for the methods of the `Blocking_queue` class, where **1** means, "requires no permission"

and that queue must be in the "OPEN" state. The `is_closed` method is slightly more interesting. Its specification says given a pure permission to the queue it will return a choice to the client. Either the conjunction that the return value (i.e., result) is true and the queue is closed, or the conjunction that the return value is false and the queue is still open. The client can choose which branch is relevant based on the actual return value and will get the abstract state of the queue that this choice implies.

### State Invariants and Unpacking

Our system also allows programmers to verify that protocols are *implemented* correctly, or at least consistently, through a concept known as a "state invariant." State invariants are concrete predicates associated with the abstract states of an object. They are allowed to mention the fields of the object. State invariants serve two purposes: For one, they allows abstract states that are visible to clients of a class (e.g., open and closed) to be associated with concrete values for the fields of that class. In this way protocols can be verified while preserving information hiding, since clients clients need only be aware of abstract states. Additionally, state invariants provide a way to associate permissions with fields rather than just with local variables. These field permissions can then be used to satisfy pre-conditions of method calls on the same fields.

State invariants can be used to specify the implementation of the `Blocking_queue`'s invariants. For example, whenever such a queue is closed, it is required that the `elements` field (a reference to a linked list) be null and the `closed` field be true. In our example we can specify

this invariant in the following manner:

```
invariant CLOSED : elements == null ⊗ closed == true;
```

This and other state invariants have been added to the queue class in Figure 3.5.

State invariants are verified using a methodology known as "packing and unpacking." Packing and unpacking [8, 31] is important because it allows us to verify "invariants" that are occasionally, and temporarily violated. This is perfect for checking state invariants. Whenever an object is in some abstract state, the state invariant for that state must hold. However, in order to transition from one state to another, the invariants must be temporarily violated, and the unpacking process allows us to do this in a controlled manner.

In such this approach, a permission to an object can be exchanged for the state invariant of that object's current state. This process is known as unpacking. While the object is unpacked, its permission cannot be used, but the permissions and facts implied by its state invariant can be. Critically, the fields of an object can only be read from or written to when the object is unpacked. Before the object can be repacked to any state, there is a proof burden. We must be able to prove the state invariant of the new state to which the object will be packed.

```
class Blocking_queue {
  invariant STILLOPEN: elements != null ⊗ closed == false;
  invariant CLOSED: elements == null ⊗ closed == true;
  // ...
  void close() : full(this) in OPEN ⟶ full(this) in CLOSED {
    synchronized { elements = null; }
    synchronized { closed = true; }
  }
}
```

Figure 3.5: State invariant specification for the concurrent queue class

When examining this methodology, we noticed that the period of time during which a thread-shared object is unpacked is a dangerous time. During this time other threads may observe the object in an inconsistent state, a state when none of the abstract state invariants hold, unless proper thread synchronization is used. Therefore our analysis must ensure that proper synchronization *is* used when necessary so that thread-shared objects can atomically transition from one state to the next.

In the formal system presented in Section 3.3 packing and unpacking are performed through the use of two expressions, `pack` and `unpack`. While this simplifies are formal treatment, the language on which we would like to perform verification does not have these expressions. As a result, our implementation (Section 3.4) performs packing and unpacking inference.

### Can Dequeue Be Verified?

Up until this point we have presented what we believe is the clearest description of `Blocking_queue`'s specification. Notionally, each queue will have a producer thread that adds items

to the queue, and several consumer threads that remove items from the queue. This sharing pattern is a very nice match for our full and pure permissions, which capture the notion of a single writer and multiple readers. However, the meaning of a pure permission is that its owner has no right to modify the object at all. While it may be true that the consumer threads cannot change the abstract state of the queue from open to closed, they must be able to modify the queue data structure in order to remove elements. This suggests that verification of the `dequeue` method as we specified it in Figure 3.4 would be impossible.

Fortunately, the flexibility of our specification language, particularly its notion of dimensions, comes to the rescue. We can consider the queue class to consist of two separate pieces. A *protocol* half, which contains the machinery for tracking the open or closed state of the queue, and the *structure* half, responsible for holding the contents of the queue. Consumer threads are not supposed to close the queue, so they cannot modify the protocol half of the object, but they are allowed to modify the structure half of the object in order to remove objects from the queue.

This scenario can be perfectly described by our specification language thanks to state dimensions. We can define two dimensions for the queue class, protocol and structure, and map the fields relating to the open/closedness of the queue into the former dimension and the fields relating to the actual queue data into the latter dimension. The finite state machine of the structure dimension will not be very interesting. In fact it will only have one state! But because permissions to dimensions of an object can be treated separately, we can give each consumer a share (modifying) permission to the structure dimension *and* a pure permission to the protocol dimension.

Assuming that we tried to verify the implementation of the `dequeue` method and ran into this problem, we would be forced to go back to our original specification and modify it by adding a new dimension, "structure." This new dimension would reflect itself in our revised specification of the `Blocking_queue` class, shown in Figure 3.6.

### Access Permissions as Thread-Sharing

In order to determine when the state of an object could potentially be changed by another thread, we need to know which objects are shared across threads. In our system, we use access permissions as an approximation of this information. If a reference is annotated with a permission that indicates the referred object can be reached via other references, we assume that those references are held by other threads. So, if a permission indicates other modifying references exist, our analysis assumes that the object may be modified concurrently unless protected by a lock.

The idea is that access permissions, even though they describe aliasing, are an abstraction of thread-sharing. When an access permission indicates other aliases exist, we can conservatively assume that those aliases are transitively reachable from the root set of another thread. Viewed in this light, let us now reexamine each of the five permission kinds presented earlier in this section:

- unique permissions are permissions to thread-local objects. These objects can be passed from one thread to another in a linear manner.
- full permissions are permissions to objects that only one thread can modify, but many threads can read. The thread with full permission can rely on the fact that no other threads can change the state of the object.

```
class Blocking_queue {
  Blocking_queue() : 1 ⟶ unique(this) in OPEN, structure { ... }

  void enqueue(Object o) : full(this, protocol) in OPEN ⊗ share(this, structure)⊗
                           share(o) ⟶ full(this, protocol) in OPEN ⊗ share(this, structure)
  { ... }

  void enqueue_last_item(Object o) : full(this, protocol) in OPEN
        ⊗ share(this, structure) ⊗ share(o) ⟶
        pure(this, protocol) in CLOSING ⊗ share(this, structure)
  { ... }

  Object dequeue() : pure(this, protocol) in STILLOPEN ⊗ share(this, structure) ⟶
                     share(result) ⊗ pure(this, protocol) ⊗ share(this, structure)
  { ... }

  boolean is_closed() : pure(this, protocol) ⟶
                        (result = true) ⊗ pure(this, protocol) in CLOSED) &
                        (result = false) ⊗ pure(this, protocol) in STILLOPEN)
  { ... }

  void close() : full(this, protocol) in OPEN ⟶ full(this, protocol) in CLOSED
  { ... }
}
```

Figure 3.6: A revised specification of the `Blocking_queue` class which now includes the structure dimension

- immutable permissions are permissions to objects that will only ever be read. All threads can rely on this object never changing state.
- pure permissions are reading permissions to objects that another thread could potentially modify. Unless mutual exclusion is guaranteed, a thread with a pure permission must assume that the object's state could change at any moment.
- share permissions are modifying permissions to objects that could potentially be modified by a number of other threads. Again, unless mutual exclusion is guaranteed, we must assume that the object's state could change at any moment.

Given access permissions in this context, our analysis works by discarding state information for references associated with share and pure permission if the current thread does not hold a lock for those objects. This discarding of state information simulates the possible effects of concurrent modification. More technically, our analysis drops the current state of the access permission down to the guaranteed state. Since the guaranteed state is, naturally, guaranteed, the object cannot leave that state even if it is modified by another thread.

Unpacking an object may give us access to the fields of that object, and those fields may be associated with permissions that we just said are exempt from concurrent modification. But is this really true? If and object that is being unpacked is associated with pure or share permission, then multiple threads could access these unique, "thread-local" objects by traversing through

the thread-shared reference. Something must be done to remedy this unsoundness. Therefore, in order to reestablish the condition that all unique and full fields of an object could not be modified concurrently by another thread, we require that the unpacking of a pure, share, or full object be done while holding a lock on the unpacked object. Now, regardless of whether a variable is a field or local variable, our analysis only needs to discard state information for unprotected share and pure references.

The requirement that a lock must be held on an object of full, share or pure permission before unpacking may seem somewhat arbitrary, so let us further explain some of the intuition behind it. We desire to treat an object of unique permission as if it were thread-local. (Or similarly, we desire to treat an object of full permission as if it were not subject to concurrent modification.) For all unique objects, one of the three following cases must hold:

- The unique reference is on the stack (i.e., is a parameter). Parameters are not reachable directly from other threads, and the object is unaliased, therefore the object really is thread-local.
- The unique reference is associated with a field, and that field has been unpacked from another unique object. By an inductive argument, because that outer unique object is "thread-local," the newly unpacked field is as well.
- The unique reference is associated with a field, and that field has been unpacked from a non-unique object. While the outer object is not thread-local, by rule the lock on the object must be held before it can be unpacked. Therefore, the lock establishes temporary thread-locality. This situation is illustrated in Figure 3.7.

So as a result, our decision to treat unique objects as thread-local is well-founded.

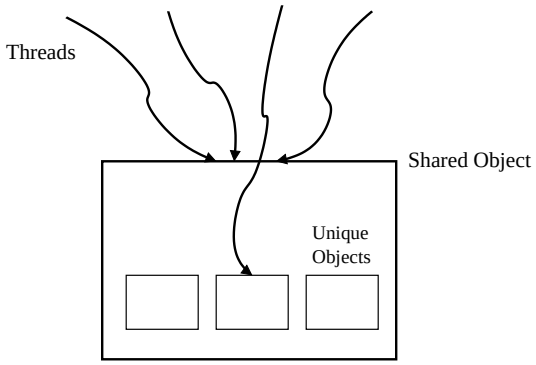


Figure 3.7: Unique and full fields within a thread-shared object have necessarily been unpacked while holding the lock. The single thread inside is free to modify at will.

Holding a lock when unpacking full, share and pure permissions helps to reestablish the thread-locality of unique permissions. But is has another benefit. It also ensures that state transitions are performed atomically, so that inconsistent concrete states, not corresponding to any abstract state, will not be visible by any other threads. This is important for preventing the sort of atomicity violation illustrated back in Figure 3.2.

Finally, we require that static member variables must be read or written to while synchronized on the associated class object. (e.g., `Blocking_queue.class`). Our formal system (Section 3.3)

does not actually have a notion of static member variables, so this restriction is not enforced. However, in our implementation, Sync-or-Swim, we do enforce this requirement.

In summary, the following additions are required to make access permissions function as a sound approximation of thread-sharing:

- We immediately forget state information about references whose access permission indicates that the referred object could be modified by other threads (pure and share).
- We require that share, pure, and full references are only unpacked when the lock is held for that reference. This ensures that we have exclusive access to the fields of that object.
- All static fields must be read from and written to inside of synchronized blocks.

### 3.2.3 Tracking Held Locks

Finally, the last part of our methodology involves mutual exclusion. The protocols on a thread-shared, modifiable object *can* be obeyed as long as proper mutual exclusion is used. In order to keep the locking specifications as simple as possible, we enforce a simple locking discipline based on synchronized blocks. Our system limits locking flexibility but requires no additional specifications beyond what is required for a single-threaded analysis.

The locking discipline enforced is simple: each object must be used as the lock to protect the abstract state of that object. Clients of a class must synchronize on instances in order to ensure that they are not modified by other threads. In order for this to work, the receiver reference (i.e., *this*) must be used for synchronization when implementing a class. This is a simple protocol, but it appears to be quite commonly used. Numerous existing approaches allow programmers to specify which locks protect which pieces of state [43, 50, 64], but they require additional specification. Still, we believe that these approaches are, in principle, compatible with our system.

In order to track whether a particular lock is held at some point in the program, we use a simple type and effect system. Typechecking is performed under a lock context, $\Psi$. Every time a synchronized block is entered, the reference on which the synchronization is occurring is added to the lock context. When the block is exited, the reference is removed from the lock context. When typechecking a method we assume that no locks are held at all. This is compatible with Java's synchronized blocks, which are reentrant locks, but is conservative.

Variables that are used as locks should not be reassigned because otherwise the locking context might become inconsistent with the set of locks actually held by the running thread. In our formal system this is not an issue, since programmers can only synchronize on local variables which can never be reassigned. However, in our implementation we must add the additional restriction that all synchronized variables are declared as final.

### 3.2.4 Verifying Our Examples

Now that we have seen typestate specifications, access permissions and we can statically track whether or not code is executing while holding a particular lock, we can revisit our original examples and check them.

Figure 3.8 contains the code from the producer thread first originally shown in Figure 3.1. The original producer thread worked correctly and here we can show that our approach has the

```
1  final Blocking_queue queue = new Blocking_queue();
2  ∅ | unique(queue, alive) in OPEN, structure
3  // ... Thread creation, see Figure 3.9
4  ∅ | full(queue, protocol) in OPEN ⊗ share(queue, structure)
5  for( int i=0;i<5;i++ )
6     ∅ | full(queue, protocol) in OPEN ⊗ share(queue, structure)
7     queue.enqueue("Object " + i);
8     ∅ | full(queue, protocol) in OPEN ⊗ share(queue, structure)
9
10 queue.close();
11 ∅ | full(queue, protocol) in CLOSED ⊗ share(queue, structure)
```

Figure 3.8: Verification of the producer thread proceeds normally. Because the producer has full permission, the analysis knows that other threads cannot close the queue.

power to verify this example. Each line of the method is annotated with the set of locks held (here always $\emptyset$) and the set of access permissions held at that point. On line 2 immediately after the call to the queue's constructor, the producer has a unique permission to the newly created queue. It must give pure permission to the consumer thread, but because of our splitting rules, it holds onto a full permission, which indicates exclusive modifying right. Even though no locks are held, this exclusive modifying right guarantees that the queue remains open. This permission satisfies the pre-condition of the enqueue call on line 7, the post-condition of which allows subsequent calls to the same method. Finally, the close method is called, leaving an access permission to the queue in the "CLOSED" state on line 11.

So we have shown that our approach can verify the consumer thread, but can it also find the race condition on the abstract state of the queue in the consumer thread? Figure 3.9 shows the use of our approach to detect this race. As in the previous example, we have annotated certain lines with the locks and access permissions known to be held at that point. The code from this example has changed somewhat since Figure 3.1. We have changed what was an anonymous class into a traditional class so that the state invariant on the queue reference could be specified. Line 10 begins with the consumer holding a unique permission to the receiver, as dictated by the method's pre-condition. This is immediately exchanged for the state invariant of the alive state on the next line through the process of unpacking. This invariant gives us a pure permission to the queue, which is enough to call the is_closed method. Line 13 shows an intermediate state of the analysis where, based on the return value of the is_closed method we know that the queue is not closed. However, by line 14 the analysis has discarded this information, dropping the permission down to the root (or guaranteed) state, alive. This is done because the pure permission indicates the possibility of other, modifying threads. On line 15 the analysis signals an error, since the pre-condition of the dequeue method has not been satisfied.

In Figure 3.10 we have taken the same consumer thread code and rewritten it to use proper synchronization. This code does not suffer from the same race condition. Since the lock on the queue is held at line 8 the state of the queue is maintained, thus satisfying the pre-condition of

```
1  class ConsumerThread extends Thread {
2    invariant alive: pure(queue, protocol) ⊗ share(queue, structure);
3    final Blocking_queue queue;
4
5    ConsumerThread(BlockingQueue q) : pure(q, protocol) ⊗ share(q, structure)
6                                      ⟶ unique(this) in alive
7    { this.queue = q; }
8
9    public void run() : unique(this) ⟶ unique(this) {
10     ∅ | unique(this)
11     ∅ | pure(queue, protocol) ⊗ share(queue, structure)
12     while( !queue.is_closed() )
13       ∅ | pure(queue, protocol) in STILLOPEN ⊗ share(queue, structure)
14       ∅ | pure(queue, protocol) in alive ⊗ share(queue, structure)
15       System.out.println("Got object: " + queue.dequeue()); // Error!
16   }
17 }
18 (new ConsumerThread(queue)).start();
```

Figure 3.9: Our approach detects the race condition in the consumer thread on line 15 because the pre-condition of the dequeue method is not satisfied.

the dequeue method.

Finally, our approach detects that implementation of the queue's close method does not atomically transition from the open state to the closed state. This method was originally given in Figure 3.2. As in the consumer thread, in this example we start out by exchanging the permission given by the method's pre-condition (line 2) for the predicate implied by the "OPEN" state (line 4). This unpacking process must be done while the lock on the receiver variable is held, but since the locking context indicates it is held, everything is okay. However, it is on line 7 where the problem is detected. On this line, the lock is released, but the current state of the context cannot be used to prove the "CLOSED" state invariant or any other state invariant. This is an error, as our approach has discovered a part of the code where an intermediate object state can be observed by other threads.

All of these figures elide certain details. In order to ensure that reentrant method calls see objects in consistent states, we are required to pack before method calls when object reentrancy is possible. Additionally, our analysis keeps track of the permissions associated with unpacked objects so that it knows what permissions to give back when those objects are packed again.

In the introduction we say that race conditions are prevented up to the program behavior that is specified, and now hopefully it is clear why. Only those method behaviors and class invariants that can be expressed in terms of typestate, and that are actually annotated by the programmer will be guaranteed in the face of concurrency.

```
1  public void run() {
2    ∅ | unique(this)
3    ∅ | pure(queue, protocol) ⊗ share(queue, structure)
4    while(true) {
5      synchronized(queue) {
6        queue | pure(queue, protocol) ⊗ share(queue, structure)
7        if(!queue.is_closed())
8          queue | pure(queue, protocol) in STILLOPEN ⊗ share(queue, structure)
9          System.out.println("Got object: " + queue.dequeue()); // OK
10       else return;
11     }
12   }
13 }
```

Figure 3.10: In this corrected version of the consumer thread, the race condition has been eliminated.

```
1  void close() : full(this, protocol) in OPEN ⟶ full(this, protocol) in CLOSED {
2    ∅ | full(this, protocol)
3    synchronized(this) {
4      this | elements != null ⊗ closed == false
5      elements = null;
6      this | elements == null ⊗ closed == false
7    } // Error!
8    synchronized(this) { closed = true; }
9  }
```

Figure 3.11: Our analysis detects that the abstract state of the queue does not transition atomically on line 7 when the synchronized block ends but the receiver remains unpacked

## 3.3 Language

We have formalized our analysis as a core, Java-like language. In this section we will present this formal language. It is presented through a series of judgments defined using inference rules. In all cases, these judgments are defined to be the strongest ones closed under the given rules.

Our formal language builds heavily upon a few existing systems in the literature. Our basic type system for checking protocols is built on the work developed by Bierhoff and Aldrich [15]. That work was an extension of Boyland [25] and Zhao's [100] work on fractional permissions. Our permission system is more expressive than this work (for example, full, pure, and share are not part of their work). The syntax and semantics for the Java core is inspired by Featherweight Java [63].

This section contains quite a bit of technical detail. Figure 3.12 presents a brief outline of the remainder of Section 3.3. In this outline we briefly describe each sub-section and the most im-

portant judgments in the language. For completeness, we have included all of the rules needed to define our language. However, many of its features are directly reused from Bierhoff and Aldrich [15]. Therefore, in our outline we highlight the judgments that have changed significantly in order to make our language sound in the face of concurrency.

| Section | Content | Relation to [15] | Description |
|---|---|---|---|
| 3.3.1 | Language Syntax | Reused | The basic syntax of the language, inspired by Featherweight Java [63]. |
| 3.3.2 | Permission Syntax | Reused | The full syntax for permissions, which is more complex and consequently more flexible than presented earlier in the chapter. |
| | Fractions | Reused | Fractional permissions [25] are added to the permission syntax. |
| 3.3.3 | Permission Rules | Mostly Reused | Rules for manipulating permissions and establishing that they are well-formed. |
| | State-space judgments | Reused | Rules for relating the various states and dimensions in a state hierarchy. |
| | $\Gamma \vdash P$ wf | Reused | A judgment defining well-formed permission predicates. |
| | $P \Rrightarrow P'$ | Reused | A judgment allowing the splitting and joining of permissions. |
| | $\downarrow^{\Psi} (P)$ | **New** | A judgment designed to weaken permissions in the face of concurrent access, given a set of held locks. |
| | $\Delta \vdash P$ | Reused | A judgment for proving permission predicates using Affine Logic. |
| 3.3.4 | Type-Checking | Modified | The rules for type-checking programs and expressions. |
| | $\Gamma \vdash t : T$ | Reused | A judgment for type-checking terms, which are simple, effect-free expressions. |
| | Helpers | Reused | A variety of helper functions and judgments needed for type-checking. |
| | $\Gamma; \Delta; \Psi; u \vdash^{C} e : E$ | **Heavily Modified** | The expression type-checking judgment, which depends upon the earlier judgments and functions. |
| | Program Well-Formedness | Mostly Reused | Rules for ensuring that complete programs are legal. |

Figure 3.12: A brief summary of the formal definition of our language

### 3.3.1 Basic Language Syntax

The syntax of this language is given in Figure 3.13. Programs consist of a list of class definitions, $CL$, and a "main" expression. Each class consists of lists of field declarations, $F$, state declarations, $R$, a constructor, $I$, lists of state invariant declarations $N$, and methods $M$. Fields, in addition to having a name and a type, are *mapped* into a node in the state hierarchy. A node, $n$, is a state or a dimension. This allows programmers to state that some field is a conceptual member of a state or dimension, and that field will be unmodifiable unless the received object is unpacked to (or above) the node into which the field is mapped. (This process will be described in more detail when the rule for typechecking a field assignment is presented.)

| | | | |
|---|---|---|---|
| *program* | $PG$ | $::=$ | $\langle \overline{CL}, e \rangle$ |
| *class decls.* | $CL$ | $::=$ | $\texttt{class}\ C\ \{\ \overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\}$ |
| *field decls.* | $F$ | $::=$ | $f : T\ \texttt{in}\ n$ |
| *nodes* | $n$ | $::=$ | $s \mid d$ |
| *state decls.* | $R$ | $::=$ | $d = \overline{s}\ \texttt{refines}\ s_0$ |
| *initial state* | $I$ | $::=$ | $\texttt{initially}\ \langle P, s_1 \otimes \ldots \otimes s_n \rangle$ |
| *state inv.* | $N$ | $::=$ | $n = P$ |
| *methods* | $M$ | $::=$ | $T\ m(\overline{T\ x}) : MS = e$ |
| *method specs* | $MS$ | $::=$ | $P \longrightarrow E$ |
| *expr types* | $E$ | $::=$ | $\exists x : T.P$ |
| *terms* | $t$ | $::=$ | $x \mid \texttt{true} \mid \texttt{false} \mid t_1\ \texttt{or}\ t_2 \mid t_1\ \texttt{and}\ t_2 \mid \texttt{not}\ t$ |
| *expressions* | $e$ | $::=$ | $t \mid t.f \mid f := t \mid \texttt{new}\ C(\bar{t}) \mid t_o.m(\bar{t}) \mid \texttt{if}(t, e_1, e_2)$ |
| | | $\mid$ | $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{spawn}\ (t_o.m(\bar{t})) \mid \texttt{synchronized}\ (t)\ e$ |
| | | $\mid$ | $\texttt{unpack}(n, k, A_1)\ \texttt{to}\ (A_2)\ \texttt{in}\ e$ |
| *types* | $T$ | $::=$ | $C \mid \texttt{bool}$ |
| *predicates* | $P$ | $::=$ | $p \mid q \mid P_1 \otimes P_2 \mid P_1 \oplus P_2 \mid P_1\ \&\ P_2 \mid 1 \mid 0 \mid \top$ |
| *facts* | $q$ | $::=$ | $t = \texttt{true} \mid t = \texttt{false}$ |
| *lock context* | $\Psi$ | $::=$ | $\bullet \mid \Psi, t$ |
| *valid context* | $\Gamma$ | $::=$ | $\bullet \mid \Gamma, x : T$ |
| *linear context* | $\Delta$ | $::=$ | $\bullet \mid \Delta, P$ |
| *unpacking context* | $u$ | $::=$ | $\mathsf{p} \mid \mathsf{up}(k, n)$ |

| | | | | | |
|---|---|---|---|---|---|
| *classes* | $C$ | *fields* | $f$ | *variables* | $x, y, z$ |
| *methods* | $m$ | *states* | $s$ | *dimensions* | $d$ |

Figure 3.13: Language Syntax. $p$, $k$ and $A$ are defined in Figure 3.14

State declarations, $R$, allow programmers to define new abstract states and dimensions for classes. Dimensions must be introduced as refinements to an existing state, and consist of a list of new, mutually-exclusive states. (Well-formedness rules in Section 3.3.4 will ensure that node names are unique, and that each dimension refines an existing state.)

The constructor, $I$, is a simplified version of the Java constructor. It lets programmers define a predicate, or specification, $P$, and a conjunction of initial states. Programmers wishing to create a new instance of the class in the given states will be required to prove predicate $P$. This

predicate will be defined over fields of the class, and must be at least as strong as the state invariants of the specified initial states. The job of enforcing these requirements, along with the requirement that each initial state inhabit a separate dimension, are left to the well-formedness rules in Section 3.3.4.

State invariants, $N$, associate a state or a dimension with a predicate over the fields of the class. Method declarations, $M$, consist of several parts: a return type, a method name, a list of parameters and their types, a specification, $MS$, and an expression that is the body of the method. A method specification consists two parts, a pre-condition predicate, $P$, and a post-condition predicate $E$. The post-condition predicate is expressed in the same syntactic form as an expression type $E$. Expression types consist of a standard type $T$ and a predicate $P$, which is allowed to mention an existentially bound program variable $x$. Such an existentially bound variable is needed so that the permission predicate that results from evaluating the given expression will have a variable to mention, a syntactic requirement. In the case of a method specification, the post-condition is allowed to mention a variable representing the returned value, hence its use of the expression type syntactic form.

The language has two groups of expression forms. Terms, $t$, are expressions that do not have side-effects, while expressions, $e$, can have side-effects, and include terms. Programs must be written in let-normal form, where each effect is sequenced using the `let` expression. Most terms and expressions are quite standard. Up until this point, the syntax of the language has been identical to the language presented by Bierhoff and Aldrich [15]. Our language introduces two new expressions, the `synchronized` expression for acquiring syntactically scoped locks, and the `spawn` expression, for spawning off method calls as new threads. In our language, as in Java, any object can be used as a lock. The typing rules in Section 3.3.4 will ensure that no expressions of Boolean type are used as locks. The `unpack` expression is the most complex expression in our language. It unpacks the receiver expression (i.e., *this*) from the guaranteed node $n$ and the assumed state $A_1$ with fraction $k$, for the duration of the subexpression. At the end of this expression, the receiver is packed to the assumed state $A_2$. (A description of the syntax of fractions and assumed states follows.)

Standard types $T$ consist of class names and the Boolean type. The next two syntactic forms describe the specifications that programmers can write. Predicates $P$ can be actual permissions $p$, whose syntax is described in the next section, or they can be Boolean facts $q$. Such facts allow programmers to relate Boolean values to the states of certain objects. Predicates can also be combined using various connectives from the linear logic, including multiplicative conjunction ($\otimes$), additive disjunction ($\oplus$), and additive conjunction ( $\&$ ). Each connective has an associated unit (**1**,**0**, $\top$, respectively).

Finally, expressions in the language will be type-checked under four different static contexts: A lock context, $\Psi$, which tracks the locks held at a certain line in the program, a valid context, $\Gamma$, tracking variable types, a linear context, $\Delta$, holding temporary facts, and an unpacking context, $u$, which tracks whether or not the current method receiver is unpacked. The locking and packing contexts are also new with respect to Bierhoff and Aldrich [15], meaning that syntax-wise, the only differences are two new expressions and two new static contexts.

### 3.3.2 Permission Syntax

The syntax of the permissions themselves, $p$ in the formal syntax, are worthy of special discussion. Our permission syntax is taken from Bierhoff and Aldrich [15], and is included for completeness. While access permissions of the form, unique(q, alive) in OPEN, are useful for a basic explanation of the system, our actual permissions are a bit more flexible and, accordingly, their full syntax is more complex.

**Fractional Permissions** While it was not necessary in our running example, our approach also supports fractional permissions. Fractions [25], as used in verification, allow weaker aliasing permissions to be recombined in order to form stronger permissions. The numerical value of the fractions indicate to the analysis at what point it is guaranteed that all aliases have been eliminated. With fractions, a unique permission, for example, can be temporarily split into three share permissions, distributed to references in different parts of the program, and then be recombined into a unique permission.

In the work of Boyland [25], fractions of value of $1$ represent un-aliased permission, while fractions between $1$ and $0$ represent reading permissions. In our system fractions are interpreted differently, as we shall explain.

$$
\begin{array}{rrcl}
\textit{permissions} & p & ::= & \mathsf{access}(r, n, g, k, A) \\
\textit{references} & r & ::= & x \mid t.f \\
\textit{fraction fct.} & g & ::= & n \mapsto k \mid g/2 \mid g_1, g_2 \\
\textit{fractions} & k & ::= & 0 \mid 1 \mid k/2 \\
\textit{assumption} & A & ::= & n \mid A_1 \otimes A_2
\end{array}
$$

Figure 3.14: Full Permission Syntax

**Complete Permission Syntax** Fractions manifest themselves in the full form of our access permissions, shown in Figure 3.14. $r$ is the reference with which the permission is associated. $n$ is the root, or guaranteed node, a state or dimension which the permission is guaranteed not to leave. $A$ is the assumed state, the state below the state guarantee which the referred object currently inhabits. The assumed state can be a conjunction of states if the object inhabits multiple states in multiple dimensions. $k$ is the fraction that determines a permission's right to modify the the assumed state of the fraction. A value of $1$ means that this permission can modify the assumed state of the object, and is the only permission with the right to do so. It also has the right to introduce new state guarantees. Values between zero and one have the right to modify the assumed state but must be aware of other permissions that have that same right. (The syntax does not allow arbitrary fractions, but rather fractions of the form $\frac{1}{2^n}$. This can be done without loss of expressiveness.) A fraction value of $0$ can only read and cannot modify the assumed state.

The last piece of permission is the fraction function, $g$. The fraction function tracks the number of other permissions that have guarantees to particular nodes in the state hierarchy. It records a mapping from each node in the state hierarchy above the guaranteed node (including

the guarantee itself) to a fraction. If the guaranteed node is mapped to 1, this means that no other permissions are depending on the guaranteed state not to change, and we are free to remove the guarantee. Otherwise, the guaranteed fraction for a node must be between zero and one, in which case the guarantee cannot be removed until all fractions pointing to that node can be recombined. The syntax allows certain nonsensical permissions to be written by the programmer. All these malformed permissions will be prevented by our language's well-formedness rules, presented in Section 3.3.3.

Now we can define the earlier permission kinds in terms of this new syntax:

$$\begin{aligned}
&\mathsf{unique}(r, n, g)\ \texttt{in}\ A \equiv \mathsf{access}(r, n, \{g, n \mapsto 1\}, 1, A)\\
&\mathsf{full}(r, n, g)\ \texttt{in}\ A \equiv \mathsf{access}(r, n, g, 1, A)\\
&\mathsf{share}(r, n, g, k)\ \texttt{in}\ A \equiv \mathsf{access}(r, n, g, k, A) \quad (0 < k < 1)\\
&\mathsf{pure}(r, n, g)\ \texttt{in}\ A \equiv \mathsf{access}(r, n, g, 0, A)
\end{aligned}$$

Notably, a unique permission is just a permission whose root, or guaranteed node, maps to 1 in the fraction function. For simplicity, we will leave the immutable permission kind out of our formal treatment, but it can be represented by adding an additional flag to distinguish share and immutable permissions.

### 3.3.3 Permission Manipulation and Well-Formedness

In this section we present the rules for checking that permissions are well-formed as written, along with the various judgments for proving and exchanging permissions. Except as noted, these judgments are due to Bierhoff and Aldrich [15].

**Permission Well-Formedness** There are several judgments that are used to ensure that permissions are well-formed. First, there are a number of state-space judgments and functions, presented in Figures 3.15 and 3.16. Broadly, these judgments ensure that the states and dimensions (together, "nodes") mentioned in permissions are related appropriately, as substates or orthogonal states, depending on the context. These judgments are needed for later well-formedness and type-checking rules.

$$\begin{aligned}
\mathsf{refinements}(\texttt{Object}) &= \bullet\\
\mathsf{refinements}(C) &= \overline{R} \quad \text{where } \overline{R} \text{ are the state refinements defined in } C
\end{aligned}$$

Figure 3.15: The state refinement function

We will go through each judgment in turn. The refinements helper function is used to look up the state refinement declarations in a given class. The judgment, $C \vdash A\ \mathsf{wf}$, establishes that a state assumption is well-formed, which is the case when all of the conjoined nodes are defined in the current class. The next series of rules defines a binary operator $\leq$ on nodes, which holds when one node is a sub-state or sub-dimension of the other. The $\#$ binary operator on state assumptions defines a notion of orthogonality between states and dimensions. Node are

$$\frac{n \; \textit{in} \; \mathsf{refinements}(C)}{C \vdash n \; \mathsf{wf}} \qquad \frac{C \vdash A_1 \; \mathsf{wf} \qquad C \vdash A_2 \; \mathsf{wf}}{C \vdash A_1 \otimes A_2 \; \mathsf{wf}} \qquad \frac{d = \overline{s} \; \texttt{refines} \; s \in \mathsf{refinements}(C)}{C \vdash s_i \leq d \qquad C \vdash d \leq s}$$

$$\frac{C \vdash n \; \mathsf{wf}}{C \vdash n \leq n} \qquad \frac{C \vdash n \leq n'' \qquad C \vdash n'' \leq n'}{C \vdash n \leq n'}$$

$$\frac{d = \overline{s} \; \texttt{refines} \; s \in \mathsf{refinements}(C) \qquad d' = \overline{s'} \; \texttt{refines} \; s \in \mathsf{refinements}(C) \qquad d \neq d'}{C \vdash d \# d'}$$

$$\frac{C \vdash n_1 \leq n_1' \qquad C \vdash n_1' \# n_2' \qquad C \vdash n_2 \leq n_2'}{C \vdash n_1 \# n_2} \qquad \frac{C \vdash A' \# A}{C \vdash A \# A'}$$

$$\frac{C \vdash A_1 \# A \qquad C \vdash A_2 \# A}{C \vdash (A_1 \otimes A_2) \# A} \qquad \frac{C \vdash n' \leq n}{C \vdash n' \prec n} \qquad \frac{C \vdash A_{1,2} \prec n \qquad C \vdash A_1 \otimes A_2 \; \mathsf{wf}}{C \vdash (A_1 \otimes A_2) \prec n}$$

$$\frac{C \vdash A \prec n \qquad \forall n' : C \vdash A \prec n' \Rightarrow n \leq n'}{C \vdash A \ll n} \qquad \frac{d = \overline{s} \; \texttt{refines} \; s \in \mathsf{refinements}(C)}{C \vdash s_i \lessdot d \qquad C \vdash d \lessdot s}$$

$$C \vdash \mathsf{nodes}(n, n) = n \qquad \frac{C \vdash n \lessdot n' \qquad C \vdash \mathsf{nodes}(n', n'') = \overline{n}}{C \vdash \mathsf{nodes}(n, n'') = \overline{n}, n'} \qquad \frac{}{C \vdash \mathsf{nodez}(n, n) = \bullet}$$

$$\frac{C \vdash n \lessdot n'}{C \vdash \mathsf{nodez}(n, n') = \bullet} \qquad \frac{C \vdash \mathsf{nodez}(n'', n') = \overline{n} \qquad C \vdash n \lessdot n'' < n'}{C \vdash \mathsf{nodez}(n, n') = n'', \overline{n}}$$

Figure 3.16: State-space judgments and functions

orthogonal if they are different dimensions of the same state, or if they are sub-nodes of such dimensions. The judgment, $C \vdash A \prec n$, defines a binary operator between state assumptions and nodes that is roughly analogous to $\leq$. It states that every node that is mentioned in the state assumption is a sub-state or sub-dimension of the right-hand node. The judgment $C \vdash A \ll n$ says that $n$ is the lowest node in the state hierarchy that is above every node mentioned in $A$. In the same spirit, the judgment, $C \vdash n \lessdot n'$, says that $n$ is an immediate child node of $n'$. Finally, two functions generate lists of nodes in a state hierarchy. The function $\mathsf{nodes}(n, n')$ generates a list of all the ancestor nodes of $n$, ending at $n'$. The function $\mathsf{nodez}(n, n')$ performs the same operation, but does not include the final ancestor $n'$ in the returned list.

Many of the state space judgments are used in the rules for permission well-formedness, shown in Figure 3.17. The most important of these judgments is, $\Gamma \vdash P \; \mathsf{wf}$. The conjunction, unit and fact rules are trivial. The permission rule, WF-PERM, is not. For each permission, the reference must be of class type and in scope. The assumed state $A$ must be below the guaranteed node $n$, and the fraction function $g$, must be well-typed and must mention every node between the guaranteed node and the root state alive. Fraction functions are well-typed as long as the

$$\frac{k_i \neq 0}{\Gamma \vdash \overline{n \mapsto k} : \overline{n} \to \mathsf{Fract}} \qquad \frac{\Gamma \vdash g : \overline{n} \to \mathsf{Fract}}{\Gamma \vdash g/2 : \overline{n} \to \mathsf{Fract}} \qquad \frac{\Gamma \vdash g : \overline{n} \to \mathsf{Fract} \qquad \Gamma \vdash g' : \overline{n'} \to \mathsf{Fract}}{\Gamma \vdash g, g' : \overline{n}, \overline{n'} \to \mathsf{Fract}}$$

$$\text{WF-PERM}\ \frac{\Gamma \vdash r : C \qquad C \vdash A \prec n \qquad \Gamma \vdash g : \overline{n} \mapsto \mathsf{Fract} \qquad C \vdash \mathsf{nodes}(n, \mathsf{alive}) = \overline{n}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \mathsf{wf}}$$

$$\frac{\Gamma \vdash P_1\ \mathsf{wf} \qquad \Gamma \vdash P_2\ \mathsf{wf} \qquad \mathsf{op} \in \{\otimes, \oplus, \,\&\,\}}{\Gamma \vdash P_1\ \mathsf{op}\ P_2\ \mathsf{wf}} \qquad \frac{\mathtt{unit} \in \{\mathbf{1}, \mathbf{0}, \top\}}{\Gamma \vdash \mathtt{unit}\ \mathsf{wf}} \qquad \frac{\Gamma \vdash t : \mathtt{bool}}{\Gamma \vdash t = \mathtt{true}\ \mathsf{wf}}$$

$$\frac{\Gamma \vdash t : \mathtt{bool}}{\Gamma \vdash t = \mathtt{false}\ \mathsf{wf}}$$

Figure 3.17: Permission well-formedness judgments

fraction for each node is greater than zero.

**Splitting, Joining and Forgetting** Now that we have well-formed permissions, we can do interesting things with them. The next series of rules define splitting and joining, which allow permissions of one type to be soundly exchanged for permissions of another type, and forgetting, an important process that will ensure that our type system does not maintain information about objects that are subject to concurrent modification.

The complete rules for splitting and joining permissions, and also for adding and removing state guarantees, are given in Figure 3.18. These rules are due to Bierhoff and Aldrich [15]. They define a judgment, $P \Rrightarrow P'$, by which permissions can be split and joined.

The SYM rule enables unique, full and share permissions to be converted into a pair of share permissions, and enables pure permissions to be converted into a pair of pure permissions. (Note that in our formal system, fraction values of zero divided by two are implicitly equivalent to zero. This simplifies the presentation.) The ASYM rule enables unique and full permissions to be converted into a full permission plus a pure permission, a share permission into a share and a pure, and a pure permission into two pure permissions. For both rules, the reverse joining operation is also permitted.

The next two rules allow full permissions to two assumed states (and by definition unique ones as well) to be split into two full permissions to those assumed states, with orthogonal guaranteed nodes. Such a rule is needed for our queue example, so that a full permission to the queue in the OPEN and structure states can be converted into two guaranteed full permissions, one for the protocol dimension and one for the structure dimension. The resulting permissions can then be joined back together, if desired, using rule F-JOIN$\otimes$.

Rule F-DOWN allows full permissions to introduce new state guarantees, while the F-UP rule allows unique permissions to drop them. A pure permission can always drop a guaranteed node, as indicated by rule P-UP. Finally, it is always safe to drop the assumed state of a permission to the guaranteed state, as indicated by rule DISCARD.

SYM

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Leftrightarrows \mathsf{access}(r, n, g/2, k/2, A') \otimes \mathsf{access}(r, n, g/2, k/2, A'')}$$

ASYM

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r, n, g, k, A) \Leftrightarrows \mathsf{access}(r, n, g/2, k, A') \otimes \mathsf{pure}(r, n, g/2, A'')}$$

F-SPLIT $\otimes$

$$\frac{n_1 \# n_2 \qquad A_1 \prec n_1 \leq n \qquad A_2 \prec n_2 \leq n \qquad p_i = \mathsf{full}(r, n_i, \{g/2, \mathsf{nodez}(n_i, n) \mapsto 1/2, n_i \mapsto 1\}, A_i)}{\mathsf{full}(r, n, g, A_1 \otimes A_2) \Rrightarrow p_1 \otimes p_2}$$

F-JOIN $\otimes$

$$\frac{\begin{array}{c} n_1 \# n_2 \\ A_1 \prec n_1 \leq n \qquad A_2 \prec n_2 \leq n \qquad p_i = \mathsf{full}(r, n_i, \{g/2, n \mapsto 1/2, \mathsf{nodez}(n_i, n) \mapsto 1/2, n_i \mapsto 1\}, A_i) \end{array}}{p_1 \otimes p_2 \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A_1 \otimes A_2)}$$

F-DOWN

$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n, g, A) \Rrightarrow \mathsf{full}(r, n', \{g, \mathsf{nodes}(n', n) \mapsto 1\}, A)}$$

F-UP

$$\frac{A \prec n' \leq n}{\mathsf{full}(r, n', \{g, n \mapsto 1, \mathsf{nodes}(n', n) \mapsto 1\}, A) \Rrightarrow \mathsf{full}(r, n, \{g, n \mapsto 1\}, A)}$$

P-UP

$$\frac{n' \leq n}{\mathsf{pure}(r, n, \{g, \mathsf{nodes}(n', n) \mapsto \bar{k}\}, A) \Rrightarrow \mathsf{pure}(r, n', g, A)}$$

DISCARD

$$\mathsf{access}(r, n, g, k, A) \Rrightarrow \mathsf{access}(r, n, g, k, n)$$

Figure 3.18: Splitting and joining of access permissions

The "forgetting" judgment, which is primarily responsible for making our analysis sound in the face of concurrent modification, is shown in Figure 3.19. This judgment is new to the language. It is the responsibility of this judgment to ensure that, in the local context, our analysis does not depend on any state of an object that could be modified concurrently by another thread. For our purposes, this specifically refers to the share and pure permissions. The first three cases of the forgetting judgment are the most interesting. To paraphrase, if we perform the forgetting process on a permission to a reference for which we hold a lock, the permission is unchanged. If we perform forgetting on a permission whose $k$ fraction is 1 (i.e., unique or full) then the permission is unchanged. (The same is true for the immutable permission, as modeled in the formal system in the next chapter.) But, if we perform forgetting on a permission whose $k$ fraction is less than 1, a share or pure permission, and the lock is not held for the associated reference, the analysis must drop the assumed state $A$ down to the guaranteed state $n$. This judgment as a result simulates the effect of concurrent modifying threads. The rest of the rules perform forgetting on more complex predicates. The final rule enacts the forgetting process on an entire linear context, $\Delta$. This judgment will be used in the typing rule for the `let` expression,

$$\downarrow^{\Psi,r} (\mathsf{access}(r,n,g,k,A)) = \mathsf{access}(r,n,g,k,A)$$

$$\frac{k < 1 \qquad r \notin \Psi}{\downarrow^{\Psi} (\mathsf{access}(r,n,g,k,A)) = \mathsf{access}(r,n,g,k,n)}$$

$$\downarrow^{\Psi} (\mathsf{access}(r,n,g,1,A)) = \mathsf{access}(r,n,g,1,A) \qquad \frac{P = q \mid \mathbf{1} \mid \mathbf{0} \mid \top}{\downarrow^{\Psi} (P) = P}$$

$$\frac{\downarrow^{\Psi} (P_1) = P_1' \qquad \downarrow^{\Psi} (P_2) = P_2'}{\downarrow^{\Psi} (P_1 \otimes P_2) = P_1' \otimes P_2'} \qquad \frac{\downarrow^{\Psi} (P_1) = P_1' \qquad \downarrow^{\Psi} (P_2) = P_2'}{\downarrow^{\Psi} (P_1 \oplus P_2) = P_1' \oplus P_2'}$$

$$\frac{\downarrow^{\Psi} (P_1) = P_1' \qquad \downarrow^{\Psi} (P_2) = P_2'}{\downarrow^{\Psi} (P_1 \,\&\, P_2) = P_1' \,\&\, P_2'} \qquad \frac{\downarrow^{\Psi} (P) = P'}{\downarrow^{\Psi} (\Delta, P) = \downarrow^{\Psi} (\Delta), P'}$$

Figure 3.19: The 'forgetting' judgment

which allows us to create sequential programs. In our typing rules the forgetting operation will occasionally be performed using an empty locking context, $\downarrow^{\bullet}$. Such an operation is meant to make a permission or a permission context sound under the assumption that no locks are held.

**Proving Permissions** Method and state invariant predicates, $P$, are written and proved in an intuitionistic affine logic. Like a linear logic, affine logics treat facts as resources that cannot be duplicated, but they additionally add the principle of weakening, which means that all resources are not required to be used. Throughout the typing rules, we will use the affine logic proof judgment, $\Delta \vdash P$, extensively. This judgment can be read as, "in the context of a list of consumable resources, the predicate $P$ can be proven true." The rules defining this judgment are standard, and given in Figure 3.20. Still, a few things are worthy of note. First, facts and permissions are treated as atomic predicates, and proven using rule LINHYP. Additionally, at any time the judgment $P \Rrightarrow P'$ can be used to split or join a permission under proof, as given by rule SUBST.

### 3.3.4 Type-Checking and Program Well-Formedness

In this section we finally have built up enough machinery in order to present the type-checking rules. These rules collectively show how our language works to prevent protocol violations as checking time.

**Term Type-Checking** First, our language has basic rules for type-checking terms, shown in Figure 3.21. The term-checking judgment, $\Gamma \vdash t : T$, shows how a type can be generated for a given term. These types do not contain any permission information, and therefore the rules are quite simple.

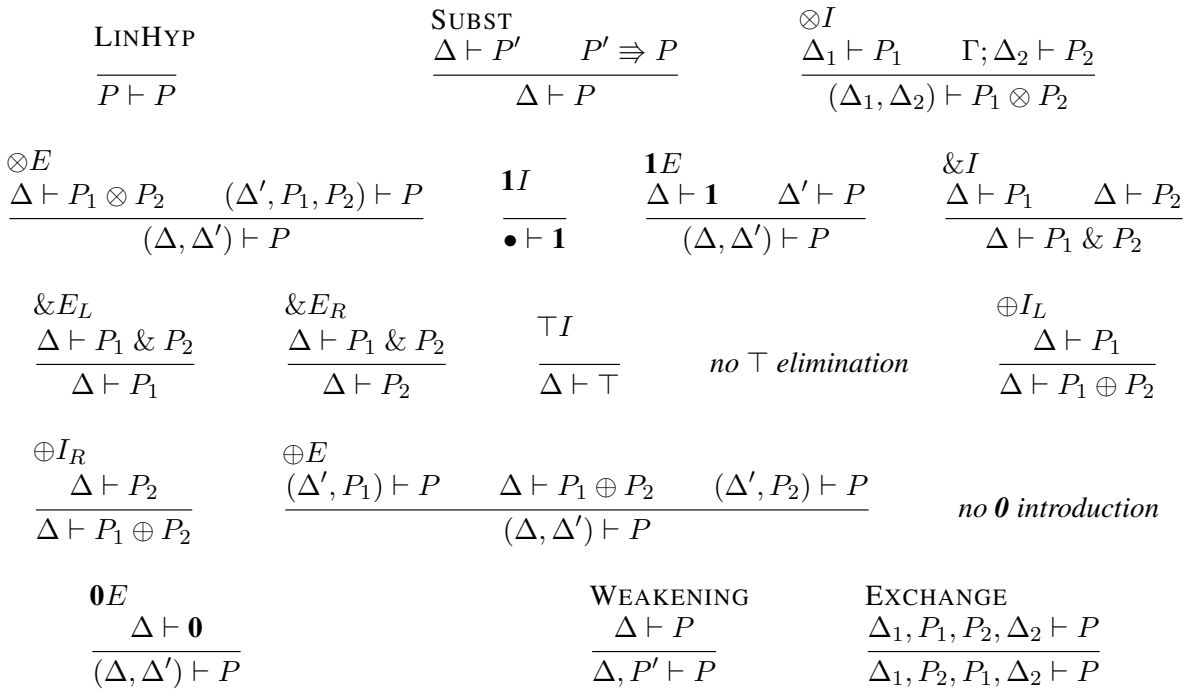


Figure 3.20: The affine logic proof judgment

**Helper Judgments for Type-Checking** The rules for type-checking expressions depend on a host of minor judgments and functions. These auxiliary judgments are needed to look up various facts about the types under discussion and perform various simple operations. They are described in Figure 3.22. All helper judgments are due to Bierhoff and Aldrich [15].

The first group of rules in Figure 3.22 are all devoted to looking up state invariants for the states and dimensions in a class, and doing so in a sound manner. The critical function is $\mathsf{inv}_C(n, g, k, A) = P$, which is used by the expression checking rules to look up a predicate associated with a state invariant. Its definition depends on the proceeding functions and judgments. The function $\mathsf{purify}(p) = P$ is used to "purify" predicates, meaning to turn all modifying per-

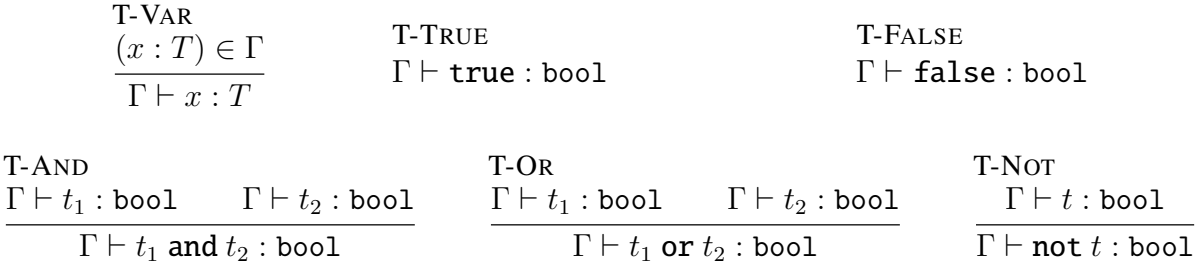


Figure 3.21: Term typechecking rules

$$\frac{p = \mathsf{access}(r, n, g, k, A)}{\mathsf{purify}(p) = \mathsf{access}(r, n, g, 0, A)} \qquad \frac{\mathtt{unit} \in \{\mathbf{1}, \mathbf{0}, \top\}}{\mathsf{purify}(\mathtt{unit}) = \mathtt{unit}}$$

$$\frac{\mathsf{purify}(P_1) = P_1' \qquad \mathsf{purify}(P_2) = P_2' \qquad \mathsf{op} \in \{\otimes, \oplus, \,\&\,\}}{\mathsf{purify}(P_1 \ \mathsf{op} \ P_2) = P_1' \ \mathsf{op} \ P_2'} \qquad \frac{\mathtt{class}\ C\ \{\ldots n = P \ldots\} \in \overline{CL}}{\mathsf{pred}_C(n) = P}$$

$$\frac{P = \bigoplus_{n' \le n'' < n} \mathsf{pred}_C(n'')}{\mathsf{pred}_C(n', n) = P} \qquad \mathsf{inv}_C(n) = \mathbf{1} \Rightarrow n$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \qquad \mathsf{pred}_C(n_i, n) = P_i' \qquad n_1 \oplus n_2 \ll n \qquad (i = 1, 2)}{\mathsf{inv}_C(A_1 \otimes A_2) = P_1 \otimes P_1' \otimes P_2 \otimes P_2' \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A) = P \Rightarrow n'}{\mathsf{inv}_C(n, A) = P \otimes \mathsf{pred}_C(n', n) \otimes \mathsf{pred}_C(n)} \qquad \mathsf{above}_C(n) = \bigoplus_{n' : n < n' \le \mathsf{alive}} \mathsf{pred}_C(n')$$

$$\mathsf{inv}_C(n, g, k, A) = \mathsf{inv}_C(n, A) \otimes \mathsf{purify}(\mathsf{above}_C(n))$$

$$\mathsf{inv}_C(n, g, 0, A) = \mathsf{purify}(\mathsf{inv}_C(n, A) \otimes \mathsf{above}_C(n)) \qquad \frac{\mathtt{class}\ C\{\ldots \overline{F} \ldots\} \in \overline{CL}}{\mathsf{localFields}(C) = \overline{F}}$$

$$\frac{\mathtt{class}\ C\{\ldots \overline{M} \ldots\} \in \overline{CL} \qquad T_r\ m(\overline{T\ x}) : P \longrightarrow \exists \mathit{result} : T_r . P' \in \overline{M}}{\mathsf{mtype}(m, C) = \forall \overline{x : T} . P \longrightarrow \exists \mathit{result} : T_r . P'}$$

$$\frac{\mathtt{class}\ C\{\ldots \mathtt{initially} \langle P, s_1 \otimes \ldots \otimes s_n \rangle \ldots\} \qquad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n}}{\mathsf{init}(C) = \langle \overline{f : T} . P, s_1 \otimes \ldots \otimes s_n \rangle}$$

Figure 3.22: Helper judgments and functions

missions into pure ones. Such a feature is needed for two reasons. First, if an object is unpacked using a reading permission, the objects referenced by fields of that object should not be modified. Second, objects are unpacked *from* some node. All fields below that node can be modified, but fields above it cannot. This helps to ensure that objects which are guaranteed never to leave a state do not leave that state, since an object cannot be unpacked above the node it guarantees. Purification accomplishes both of these goals.

Next, the $\mathsf{pred}_C$ function, in its two forms, is used to look up state invariants from the classes that define them. $\mathsf{pred}_C(n)$ looks up the state invariant for a single node while $\mathsf{pred}_C(n, n')$ looks up and creates a conjunction of the state invariant predicates for every node between $n$ and $n'$, exclusive of $n'$.

The $\mathsf{inv}_C$ function also has several forms. Collectively they look up state invariants but in a way that carefully considers state hierarchies and permissions. The judgment $\mathsf{inv}_C(A) = P \Rightarrow n$ produces a predicate that includes the state invariants for all of the nodes mentioned in $A$, while

at the same time producing as an output the node $n$ that is immediately above the highest node in $A$. Judgment $\mathsf{inv}_C(n, A) = P$ produces a predicate that includes the state invariants for all of the nodes in $A$, for node $n$, and for all of the nodes between $A$ and $n$. Lastly, the judgment $\mathsf{inv}_C(n, g, k, A) = P$, which is the one that will actually be referenced by our typing rules, generates a predicate that includes all state invariants for the nodes of the object, from $A$ up to the alive state. If the permission is a pure one, the state invariants for the nodes between the guaranteed and the alive state will be purified.

Finally, the remaining functions are used for convenient look-up of facts related to class definitions. $\mathsf{localFields}(C)$ returns a list of the fields defined by class $C$. $\mathsf{mtype}(m, C)$ returns the specification of method $m$ defined in class $C$. $\mathsf{init}(C)$ returns the list of fields defined by the type $C$, along with the constructor predicate that must be proven in order to construct instances of $C$ and the states that the newly constructed object will be in. All of this information will be useful when type-checking object instantiations.

**Expression Type-Checking** Expressions are type-checked using the following judgment: $\Gamma; \Delta; \Psi; u \vdash^C e : E$. The rules defining the judgment are presented in Figures 3.23 and 3.24. This judgment says, "given a list of variable types that can be used many times, $\Gamma$, and a list of consumable predicates that can be used only once, $\Delta$, and a context of currently held locks, $\Psi$, an unpacking context $u$, the expression $e$ being executed within receiver class $C$ has type $E$." Note that for clarity of presentation the receiver class annotation is left off unless it is needed in a typing rule. The unpacking context is somewhat unusual. It tracks statically whether or not the object receiver of the current method (i.e., *this*) is unpacked at the expression being checked. Since unpacking is syntactically scoped, tracking this fact is not a particularly difficult process. However, the unpacking context also keeps track of the root state to which the object was unpacked, as well as the fraction of permission with which the receiver was unpacked. This information will be used to determine which fields, if any, can be legally assigned.

Expression types $E$ consist of a standard type $T$ and a permission $P$. This permission may contain existentially bound variables. The existential type of an expression is somewhat unusual and therefore deserves further mention. The reason a permission can contain existentially bound variables is because, while normally a permission is associated with a reference, there are times when our system tracks the permissions of an object to which no reference points. For instance, after the first sub-expression of a let binding is evaluated, the result (if of a class type) is an object, and before it is bound to a variable, the available permission to this object must be tracked. Similarly, after a field has been reassigned, the permission to the object to which it previously referred still exists and can be reassigned to another reference. In rule P-ASSIGN, one can see this process occurring in the resulting permission $[f_i/x]P$, where the field to which object is assigned, $f_i$, is being substituted in for the bound variable $x$. Thus, giving expressions existential types allows us to keep consistent object permissions and the references that point to those objects.

The declarative nature of the affine logic judgment can make for typing rules that appear to come up with permissions from almost no information. See, for example, the $\Delta \vdash P$ premise of the P-TERM rule. Similarly, several typing rules divide the linear context in a seemingly arbitrary manner, written as $(\Delta, \Delta')$. In reality, the affine logic judgment works more like a constraint solver. In a typing derivation, different rules restrict the permissions or the context in

various ways, and it is the job of the implementation to find a rearrangement of permissions that satisfies all of these constraints [13, Ch. 5].

P-SYNC
$$\frac{\Gamma \vdash x : C \qquad \Gamma; \Delta; \Psi, x; u \vdash e : E}{\Gamma; \Delta; \Psi; u \vdash \texttt{synchronized}\ (x)\ e : E}$$

P-UNPACK-SYNC
$$\frac{\begin{array}{c}\Delta \vdash \mathsf{access}(\textit{this}, n, g, k, A) \\ k = 0 \Rightarrow A = A' \qquad \Gamma; \Delta', \mathsf{inv}_C(n, g, k, A); \Psi, \texttt{this}; \mathsf{up}(k, n) \vdash^C e : \exists x : T.P \\ P \vdash \mathsf{inv}_C(n, g, k, A') \otimes P_e \qquad \textit{no field perms in } P_e\end{array}}{\Gamma; \Delta, \Delta'; \Psi, \texttt{this}; \mathsf{p} \vdash^C \texttt{unpack}(n, k, A)\ \texttt{to}\ (A')\ \texttt{in}\ e : \exists x : T.P_e \otimes \mathsf{access}(\textit{this}, n, g, k, A')}$$

P-UNPACK
$$\frac{\begin{array}{c}\Delta \vdash \mathsf{access}(\textit{this}, n, (g, \{n \mapsto 1\}), 1, A) \\ k = 0 \Rightarrow A = A' \qquad \Gamma; \Delta', \mathsf{inv}_C(n, (g, \{n \mapsto 1\}), 1, A); \Psi; \mathsf{up}(1, n) \vdash^C e : \exists x : T.P \\ P \vdash \mathsf{inv}_C(n, g, k, A') \otimes P_e \qquad \textit{no field perms in } P_e\end{array}}{\Gamma; \Delta, \Delta'; \Psi; \mathsf{p} \vdash^C \texttt{unpack}(n, 1, A)\ \texttt{in}\ e : \exists x : T.P_e \otimes \mathsf{access}(\textit{this}, n, (g, \{n \mapsto 1\}), 1, A')}$$

P-CALL
$$\frac{\Gamma \vdash t_o : C_o \qquad \Gamma \vdash \overline{t : T} \qquad \Delta \vdash [t_o/\textit{this}][\overline{t}/\overline{x}]P \qquad \mathsf{mtype}(m, C_o) = \forall \overline{x : T}.P \longrightarrow \exists \textit{result} : T.P_r}{\Gamma; \Delta; \Psi; \mathsf{p} \vdash t_o.m(\overline{t}) : \exists \textit{result} : T.[t_o/\textit{this}][\overline{t}/\overline{x}]P_r}$$

P-SPAWN
$$\frac{\begin{array}{c}\Gamma \vdash t_o : C_o \\ \Gamma \vdash \overline{t : T} \qquad \Delta \vdash P' \qquad \downarrow^\bullet (P') \vdash [t_o/\textit{this}][\overline{t}/\overline{x}]P \qquad \mathsf{mtype}(m, C_o) = \forall \overline{x : T}.P \longrightarrow E\end{array}}{\Gamma; \Delta; \Psi; \mathsf{p} \vdash \texttt{spawn}\ (t_o.m(\overline{t})) : \exists \_ : \texttt{bool}.1}$$

Figure 3.23: Expression-typing rules

Now we will discuss each of the typing rules from Figures 3.23 and 3.24 in turn. Approximately half of the expression-typing rules have been reused directly from Bierhoff and Aldrich [15]. We will note when this is the case.

- P-SYNC: When typing the synchronized block, we must check that the synchronized variable has a class type. In Java only classes (not primitives) can be used as locks. Once this is done, the sub-expression is checked under a new typing context that includes the newly synchronized variable, $x$.
- P-UNPACK-SYNC: The unpack expression is broken into two rules. As discussed in Section 3.2, our system requires that when unpacking share, pure and full the associated lock must be held. In this case, a lock for the receiver (i.e., `this`) is held, as indicated by the locking context. This means that any permission kind can be unpacked. The premises add a few additional requirements. No objects are allowed to be previously unpacked ($u = \mathsf{p}$). There must also be enough permission in the linear context to prove the required receiver permission $p$. For this we defer to the linear proof judgment. The sub-expression $e$ is

P-FIELD
$$\frac{\mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad \Delta \vdash P}{\Gamma; \Delta; \Psi; \mathsf{up}(k, n) \vdash^C f_i : \exists x : T_i[x/f_i]P}$$

P-ASSIGN
$$\frac{\Gamma; \Delta; \Psi \vdash t : \exists x : T_i.P \qquad \Delta' \vdash [f_i/y]P' \qquad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad n_i \leq n \qquad k > 0}{\Gamma; (\Delta, \Delta'); \Psi; \mathsf{up}(k, n) \vdash^C f_i := t : \exists y : T_i.[f_i/x]P \otimes P'}$$

P-NEW
$$\frac{\Gamma \vdash \overline{t : T} \qquad \mathsf{init}(C) = \langle \overline{f : T}.P, A\rangle \qquad \Delta \vdash [\overline{t/f}]P}{\Gamma; \Delta; \Psi; u \vdash \mathtt{new}\ C(\bar{t}) : \exists x : C.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, 1, A)}$$

P-TERM
$$\frac{\Gamma \vdash t : T \qquad \Delta \vdash P}{\Gamma; \Delta; \Psi; u \vdash t : \exists x : T.[x/t]P}$$

P-IF
$$\frac{\Gamma \vdash t : \mathtt{bool} \qquad \Gamma; \Delta, t = \mathtt{true}; \Psi; u \vdash e_1 \exists x : T.P_1 \qquad \Gamma; \Delta, t = \mathtt{false}; \Psi; u \vdash e_2 \exists x : T.P_2}{\Gamma; \Delta; \Psi; u \vdash \mathtt{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2}$$

P-LET
$$\frac{\Gamma; \Delta; \Psi; u \vdash e_1 : \exists x : T.P \qquad \Gamma, x : T; \downarrow^{\Psi} (P, \Delta'); \Psi; u \vdash e_2 : E \qquad x \notin E}{\Gamma; (\Delta, \Delta'); \Psi; u \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : E}$$

Figure 3.24: More expression-typing rules

checked, assuming the current state invariant, $\mathsf{inv}_C$, and assuming that the receiver is unpacked ($u = \mathsf{up}(k, n)$). The resulting permission is used to prove that the object can be packed, by proving the state invariant for the new state $A'$. Any remaining permission is returned as part of the entire expression's type. Note that if the object is unpacked in a read-only state ($k = 0$), the state to which the object is packed must be the same as the state from which is was unpacked. Finally, once the object is packed, no field permissions should persist to subsequent expressions.

- P-UNPACK: This rule is similar to the P-UNPACK-SYNC rule except that it remains well-typed even if no locks are held. Accordingly, it requires a unique permission to the receiver object, signified by the fact that the permission to the root node $n$ is 1.
- P-CALL: This rule types method calls. We retain the original restriction of Bierhoff and Aldrich's system that the receiver object must be in a packed state, and note that it is generally possible to pack to an intermediate state in the event of recursive calls. The notation $[t/x]P$ signifies capture-avoiding substitution and is used throughout. It means, "replace $x$ with $t$ in $P$, alpha-converting if necessary." Before calling, we must be able to prove the pre-condition $P$ using the current linear context. The post-condition, $P_r$ is included in the resulting type of the expression. This rule is unchanged from Bierhoff and Aldrich [15].
- P-SPAWN: In our language thread spawns are very similar to method calls. A spawn is simply a method call wrapped in the `spawn` keyword. Accordingly, the requirements to

type a thread spawn are quite similar. There is one additional constraint, however. When calling a method that requires share or pure permission, the state assumptions for those permissions must be equal to the state guarantees. This we express by requiring that a permission "forgotten" with an empty locking context, $\downarrow^{\bullet} (P')$, be used to prove the pre-condition. This requirement was discovered during the process of proving type-safety. In effect it is required because the newly spawned thread may immediately race with the spawning thread or any other existing thread. Thread spawns return no permission and they ignore the return type of the spawned method.

- P-FIELD: A field read proves some permission $P$ which contains permissions for $f_i$ and existentially binds it so that it can be assigned to another reference. This rule is unchanged from Bierhoff and Aldrich [15].
- P-ASSIGN: When we assign a value to a field, we must first prove that the value has some permission and that it is the same type as the $i$th field of class $C$ to which we are assigning. The next premise requires that we prove the field currently has some permission. It must be the case that the receiver is unpacked ($u = \mathsf{up}(k, n)$). The unpacked permission must be a modifying permission, signified by $k$ being greater than zero. Additionally, the field to which we are assigning must be mapped to a node in the state hierarchy below or equal to the unpacked node. The resulting permission of the entire expression is the permission to the field's old value, suitable for assignment to another variable, as well as permission to the field's new value and the unpack predicate. This rule is unchanged from Bierhoff and Aldrich [15].
- P-NEW: In order to instantiate a new object, we must be able to prove the state invariant for the initial state of that object. This is done by looking up the state invariant $P$ for the initial state $A$, and proving it when treating the permissions to the constructor arguments as fields of the object. These permissions are consumed, and the result is a unique permission to the object in the initial state. This rule is unchanged from Bierhoff and Aldrich [15].
- P-TERM: Individual terms are given a permission and a type by type-checking the term, proving some permission $P$ from the linear context and then pulling the term itself out of the permission, resulting in an existentially bound one. This rule is unchanged from Bierhoff and Aldrich [15].
- P-IF: The conditional expression binds a Boolean term in both the branch expressions. Each branch is type-checked with the knowledge that the term is either true or false. The resulting permission for the entire expression is a disjunction, since the permission from either branch could be produced. This rule is unchanged from Bierhoff and Aldrich [15].
- P-LET: The rule for typechecking the `let` expression is important because it bears the primary responsibility for "forgetting," or discarding information that is subject to modification by other threads. In this rule the first sub-expression $e_1$ is typechecked, then the second sub-expression $e_2$ is typechecked under a new assumption. This new assumption includes a variable $x$ whose type is $T$ and that is associated with the permission that was the result of typechecking the first expression. Critically, the entire linear context under which the second expression is typechecked is "forgotten" with respect to the locking context $\Psi$. This will downgrade the state of any share or pure permissions not protected by

locks.

**Program Well-Formedness** Well-typed expressions must exist in the larger context of a well-formed program. The rules in Figure 3.25, describe program well-formedness rules, and will depend on the expression typing judgment just presented. Only rules P-PROG and P-METH-DECL have been modified from Bierhoff and Aldrich [15]. These modifications were minor, and were necessary in order that top-level expressions be typed under an empty locking context. Let us discuss each rule in turn:

$$\text{P-PROG}\quad \frac{\overline{CL}\ \mathsf{ok} \qquad \bullet;\bullet;\bullet;\mathsf{p} \vdash e : E}{\langle \overline{CL}, e\rangle : E} \qquad \text{P-CLASS}\quad \frac{\overline{F}\ \mathsf{ok\ in}\ C \ldots \overline{M}\ \mathsf{ok\ in}\ C}{\mathtt{class}\ C\ \{\ \overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\}\ \mathsf{ok}}$$

$$\text{P-FDECL}\quad \frac{f_i\ \text{unique} \qquad T_i \in \overline{CL} \cup \{\mathtt{bool}\} \qquad n_i\ \textit{declared in}\ C}{\overline{f : T\ \mathtt{in}\ n}\ \mathsf{ok\ in}\ C} \qquad \text{P-RDECL}\quad \frac{d\ \textit{unique} \qquad s_i\ \textit{unique} \qquad C \vdash s_0\ \mathsf{wf}}{d = \overline{s}\ \mathtt{refines}\ s_0\ \mathsf{ok\ in}\ C}$$

$$\text{P-CONSTR}\quad \frac{\mathtt{class}\ C\{\ldots s_1 = P_1 \ldots s_n = P_n \ldots\} \in \overline{CL} \qquad P \vdash P_1 \otimes \ldots \otimes P_n}{\mathtt{initially}\langle P, s_1 \otimes \ldots \otimes s_n\rangle\ \mathsf{ok\ in}\ C}$$

$$\text{P-SINV}\quad \frac{\mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad \overline{f : T} \vdash P_i\ \mathsf{wf} \qquad n_i\ \text{unique} \qquad \textit{for share and pure permissions, A=n}}{\overline{n = P}\ \mathsf{ok\ in}\ C}$$

$$\text{P-METH-DECL}\quad \frac{E = \exists \textit{result} : T_r.P_r \qquad \Gamma = (\overline{x : T}, \mathtt{this} : C) \qquad \Gamma \vdash P_r\ \mathsf{wf} \qquad \Gamma \vdash P\ \mathsf{wf} \qquad \Gamma; P; \bullet; \mathsf{p} \vdash^C e : E}{T_r\ m(\overline{Tx}) : P \longrightarrow E = e\ \mathsf{ok\ in}\ C}$$

Figure 3.25: Top-level well-formedness rules. Helper judgments defined in Figure 3.22.

- P-PROG: A program type-checks if all of its classes are well-formed and the single, top-level expression type-checks.
- P-CLASS: A class declaration is well-formed if its parts are well-formed.
- P-FDECL: The well-formedness rule for field declarations is somewhat informal, as are the remaining well-formedness rules. This rule states that a field declaration is well-formed if its name is unique inside the current class, if it has a proper type and if the node (state or dimension) it is declared to inhabit exists.
- P-RDECL: Programmers declare dimensions and the states that inhabit them. A dimension declaration is legal if all of the newly introduced states, along with the new dimension are uniquely named and if the state being refined has been declared for the class.

- P-CONSTR: When defining a constructor in our language, programmers may need to declare that object as being in multiple states from different dimensions. The required constructor pre-condition, $P$, must be strong enough to prove the state invariants for each of those states.
- P-SINV: A state invariant declaration is well-formed if three conditions hold. First, the state name must be unique within the current class. Next, $P$ must be well-formed, mentioning only fields of the current class. Finally, invariants describing share and pure permissions to fields cannot mention states below the guaranteed node.
- P-METH-DECL: Methods must be checked so that we can ensure the specified pre- and post-conditions are a sound description of their behavior. Given a specified pre-condition $P$ and a post-condition $P_r$, a method $m$ is checked assuming the predicate $P$ in the linear contex. The method is also checked under the assumption that no locks are held, since a method can be called from multiple dynamic locking contexts. The resulting permission returned from typechecking the expression should contain the method's post-condition.

While we do not provide dynamic semantics for this language, they are straightforward, since states and permissions are not represented at run-time. In the next chapter we present operational semantics for the core language for which we prove type safety. In that language, the dynamic semantics track as part of the heap the abstract state of each object. This is a technical requirement of the proof and allows us to show that static permissions can actually guarantee the run-time state of objects. These additions to the heap are not a requirement of an actual implementation. All permission information can be successfully erased, and need not contribute any run-time overhead.

The practical result of type safety, proved in the next chapter, is that an object at run-time will always be in the abstract state described by its static access permission even if that object is shared amongst multiple modifying threads.

### 3.3.5 Transactional Memory and Atomic Blocks

This chapter was adapted from an existing publication. That work [10] described a very similar language for preventing violations of protocols in concurrent programs that used atomic blocks as the primary means of mutual exclusion. Atomic blocks are the main primitive of mutual exclusion provided to programmers using transactional memory [53, 57, 58, 59], a recent and active area of research.

Atomic blocks provide programmers with a very simple semantics: when a thread is executing within an atomic block, it will execute *as if* no other threads are executing at the same time. Atomic blocks are typically implemented using optimistic approaches which will let threads run free and then abort and restart threads when the runtime detects they may have seen a view of memory inconsistent with the given semantics.

Here we have chosen to focus on the lock-based version of our approach, in large part to avoid repeating the full details of our own published work. Still, there is a compelling case for using lightweight verification such as our own in a transactional memory system, which we will reiterate here.

Our main point is that, atomic blocks greatly simplify the development of concurrent, shared-memory programs, although they do not eliminate all issues. Their simple semantics means that a programmer must only decide which actions should occur atomically and then specify so in a declarative manner. Contrast this with lock-based languages where a programmer must associated particular locks with particular pieces of state in order to achieve reasonable concurrency. Some transactional systems, those that provide "strong atomicity" [20], can even make data races impossible.

Still, the question remains, how does one determine which pieces of a program need to be made atomic in order to preserve the key invariants maintained by the data structures in a program? Atomic blocks do not make this process any easier. In our approach, by providing our tools with an abstract description of the invariants and conditions that must hold even in the face of concurrent access, users can be automatically told which sections of the program require atomicity. Such knowledge will be required even in a future where atomic blocks replace locks as the dominant primitive of mutual exclusion. Our original work [10] addressed just such challenges, and was proved sound in an accompanying technical report [9].

## 3.4 The Sync-or-Swim Checker

We have implemented the approach described in this chapter as a static analysis for Java programs. This tool is called, "Sync-or-Swim." Rather than attempting to define a new language with a new type system, our goal was to get the same benefits of this type system through a static analysis for an existing language. In this way we can use our approach to verify correct API use in the wealth of existing open-source Java programs. Our case studies, described in Chapter 6, we used the Sync-or-Swim tool to do exactly that.

Sync-or-Swim is a sound, modular, lattice-based static analysis in the style of "abstract interpretation," similar to many existing tools [28]. It is flow-sensitive and maintains some path-sensitivity for the purposes of tracking the results of conditionals. After each conditional, all paths are merged. Sync-or-Swim is actually a simple extension to Plural, a single-threaded protocol checker. Plural is the static protocol checker developed by Kevin Bierhoff and this author as part of a larger research project exploring protocol conformance. The Plural tool is built into the Eclipse development platform as a plug-in. In his thesis, Bierhoff devotes considerable time to explaining the implementation of the tool and the motivation of its specific features [13, chapter 6]. Therefore, here we will mainly describe the additions made to the Plural tool and some of the most important features.

Plural specifications are written using the Java annotation language. This language can express most of the features of the linear logic specification language we presented in this chapter. Receiver and parameter specifications generally take the following form:

```
@Share(requires="OPEN", ensures="CLOSED")
```

This says that a particular reference requires a share permission in the "OPEN" state, and returns a share permission in the "CLOSED" state.

Plural's specification language is quite powerful. It allows programmers to write state invariants, describe guaranteed permissions, specify dynamic state test methods, specify "borrowed"

and "captured" permissions, and allows programmers to make complex specifications relating the typestate of a class and its superclasses.

Sync-or-Swim essentially inherits this specification language wholesale, without additions or modifications. As a result, its specifications are quite expressive.

Plural associates each line of the program with a mapping from each reference to a lattice. A local, must-alias analysis helps us keep track of abstract object locations even when they are reassigned to other local variables. The lattice determines which permission is available at each program point. But inside the lattice, those permissions are represented as abstract constraints. Constraints are added when method pre-conditions consume permissions and when method post-conditions return permissions. When a method body has been completely analyzed, Fourier-Motzkin elimination is performed to solve the system of constraints [13, chapter 5]. At a high level, this process ensures that satisfying fractional values exist for each permission, without actually finding those fractions. This algorithm is exponential in the worst case.

By using a lattice-based analysis, programmers are not required to write loop invariants. At each control flow merge, the analysis goes through each reference in scope, gathers the lattice value from both merged edges, and performs a lattice join operation on those values. The result is that the minimum piece of information known from each branch is used. For example, if a unique permission is held to a variable $x$ before a loop, but on the back edge a share permission is held, these permissions will be joined, resulting in a share permission. If in each path the reference is in a different abstract state, then the lowest common ancestor state from both paths will be used.

Additionally, Plural performs packing and unpacking inference. This is important since the Java language does not have the unpack expression described in our formal language. This process is non-trivial because at each field access, the receiver may need to be unpacked from any of the states that it defines, and before method calls it may need to be packed to any of the states that it defines. In order to achieve this goal, the full lattice in Plural actually consists of a series of lattices, one for each tree of packing and unpacking choices. Every time there is a choice of states to unpack from or to pack to, a copy of the lattice will be created for each choice, and the remaining analysis process will continue in parallel for each lattice. While lattices are pruned as they become unsatisfiable, this can lead to an exponential number of lattices being introduced and tracked. Still, Plural's algorithm is decidable. We know this because the lattice itself is of finite height, and because the lattice contains a finite number of references. Eventually, in the worst case, the lattice value for every reference will be joined to one of pure permission in the alive abstract state, at which point no more join operations will be performed. (In practice, the performance of Sync-or-Swim is quite good, taking, on average, less than a second per method.)

Most of the differences between the Plural and Sync-or-Swim implementations exist in the verification routines. Sync-or-Swim modifies the verification process in two important ways. First, it carries out the process of "forgetting" between the execution of each sub-expression. As part of this process it must keep track of the locks held at each program point, a process which is performed by a simple tree-walker analysis. Second, Sync-or-Swim checks the available permission each time a receiver is unpacked or packed. If an object is unpacked with full, share or pure permission and does not hold the lock for that object, a warning will be issued. Similarly, if a lock is released for an unpacked object associated with one of these permissions then a warning will be issued. These two modifications are in large part sufficient to make the checker

sound in the presence of concurrent threads.

One additional change is required, however. Because of the way in which locks are tracked, Sync-or-Swim does not allow the reassignment of variables that are being used as locks. Therefore, the analysis requires that all variables used as the target of a synchronized block are *final*, meaning that they cannot be reassigned. Locking on non-final variables will result in a warning. The receiver variable, *this*, which is most often used as a lock, is implicitly final.

## 3.5 Related Work

This this section we describe the wealth of work related to our approach. This includes work that has either the same goals or uses very similar technology. The related work is largely divided into two groups, work that can verify behavior of concurrent programs and work that is designed to help find race conditions.

### 3.5.1 Verifying Behavior of Concurrent Programs.

As previously mentioned, this work draws heavy inspiration from Bierhoff and Aldrich [15], whose Access Permissions formed a major extension of Boyland's Fractional Permissions [25]. Bierhoff's work as developed was unsound in the face of concurrent access. Boyland's work, on the other hand, was primarily focused on preventing concurrency violations. However, his system limited programmers to unique and immutable permissions, and therefore lacked much of the flexibility of our system.

Counting permissions [22] is an approach for dealing with mutable shared memory that is in many ways similar to fractional permissions. Both systems allow writing permission to a resource to be split up to form multiple reading permissions and later recombined to form a writing permission. However, counting permissions differ in the kinds of patterns that they allow to be verified. In particular, they seem to be best suited for programs that recombine permission in an arbitrary order, as opposed to fractions which seem better suited for symmetric splitting and recombination. A good case where counting permissions are useful is in a program of multiple readers and a single writer [22]. In this example, the number of reading permissions outstanding can be made equivalent to the program's dynamic counter of readers, which allows the proof to go through. While such flexibility may be useful in some case, it was not found to be necessary during our own case studies.

The work that most closely resembles our own was developed as part of the Spec$^{\#}$ Project. Jacobs et al. [64] have also created a system that will preserve object invariants even in the face of concurrency. Moreover, our system uses a very similar unpacking methodology which comes from a shared heritage in research methodology [8]. Nonetheless, we believe our work to be different in several important ways. First, they use ownership as their underlying means of alias-control, which imposes some hierarchical restrictions on the architecture of an application. Typical ownership systems do not have a reader/writer pair such as our full and pure permissions. This makes our approach more flexible when it comes to aliasing and thread-sharing. Even ownership systems based on Universe Types [35], which account for read-only references, are more limited. In these approaches, read-only references are not guaranteed a consistent view

of the object. This means that important patterns, such as the `is_closed` method in our queue example, cannot be verified.

On the other hand, their system allows more expressive specifications, as behaviors can be specified in first-order predicate logic, rather than typestate. While we believe our approach would neatly accommodate more expressive specifications which we plan to investigate as part of future work, typestate provides a simple abstraction of object state and of effects on that object. Their system does have a proof of soundness but provides neither formal typing rules nor a formal semantics.

Their system also is restrictive in the types of objects that can be mentioned in object invariants. Once an object becomes thread-shared, a process which must be signified by the "share" annotation, it can no longer be mentioned in another object's invariant. Therefore, examples like the one shown in Figure 3.26 where the invariant of the RequestProcessor class depends on the thread-shared RequestPipe object, cannot be verified.

Finally, our system enforces a simple, common locking protocol sacrificing some flexibility. In their approach, in order to determine whether it is the responsibility of the client or provider to ensure proper synchronization, there is a notion of *client-side locking* versus *provider-side locking*. Methods using client-side locking can provide more information-laden post-conditions, while provider-side locking methods cannot. Our system forgoes this distinction, depending on the reentrant nature of Java's synchronized block. In exchange for a lower specification burden, in our system programmers may occasionally have to acquire a lock to the same object multiple times.

Some related work has also been done within the context of the JML project [84]. This work is mainly focused on introducing new specifications useful for those who would like to verify lock-based, concurrent object-oriented programs. Some of the specifications can be automatically verified, however due to the fact that this verification is done with a model-checker, verification failed to terminate on about half of their examples.

There are a number of popular logics for concurrency, which can be used to prove important properties of concurrent programs. These logics include the logic of Owicki and Gries [78], Concurrent Separation Logic [26, 77], and Rely-Guarantee Logic [66]. All three allow you to specify invariants over thread-shared, mutable data in simple imperative languages. Owicki-Gries and Concurrent Separation Logic are similar, differing in the expressive power of the logics they each use. In these systems, one associates both a lock and an invariant with a piece of thread-shared data. Upon entering a critical section, the invariants over thread-shared data are revealed. These invariants can be used to prove other propositions, but must be reestablished before the end of the critical section. This characteristic is quite similar to unpacking of state invariants in our system which, for references of full, share, and pure permission, must be performed while holding a lock. Concurrent Separation Logic furthermore allows one to reason modularly about heap memory that cannot be thread-shared, and does so in a manner that is similar to our unique permission.

Standard Separation Logic has been extended with fractions [22], and with abstract predicates [79], which provide a client/implementation abstraction layer over data structure invariants, and function much like our state invariants. If Concurrent Separation Logic were extended with both of these features, the result would be a system that is in many ways quite similar to our own. Moreover, Separation Logic is quite powerful, and therefore the the resulting system would be

```
class RequestProcessor {
  states IDLE, RUNNING;

  IDLE := full(requestPipe, closed)
  RUNNING := full(requestPipe, opened)

  RequestPipe requestPipe = new RequestPipe();

  void start() :
    unique(this, IDLE) ⟶ unique(this, RUNNING)
  {
    this.requestPipe.open();
    // Handler(rp) : pure(rp, ?) ⟶ 1
    (new Thread(new
           Handler(this.requestPipe))).start();
    (new Thread(new
           Handler(this.requestPipe))).start();
    return;
  }

  void send(String str) :
    unique(this, RUNNING) ⊗ immutable(str, default) ⟶
        unique(this, RUNNING)
  {
    this.requestPipe.send(str);
    return;
  }

  void stop() :
    unique(this, RUNNING) ⟶ unique(this, IDLE)
  {
    this.requestPipe.close();
    return;
  }
}
```

Figure 3.26: `RequestProcessor`, an example of a server-like program where class invariants depend on thread-shared objects.

able to verify a large variety of behavioral properties, rather than just object protocols.

What's more, approaches for automated verification based on Separation Logic have recently been developed. jStar [37] is a verification tool for Java programs that is based on standard Separation Logic. Smallfoot [12] is a verification tool for a custom imperative language that allows automated verification of Concurrent Separation Logic specifications.

Still, there are reasons why programmers might prefer our approach in certain situations. In practice, writing specifications in Separation Logic often involves complex specifications, which may "thread" permissions to memory cells into a variety of other method specifications that, conceptually, are not involved with the referenced cell. More concretely, it does not seem to be possible to verify our queue example with Concurrent Separation Logic. The full permission gives the producer thread the guarantee that the queue will never be closed by other threads, even though access to the queue it potentially racy and must be done under the protection of mutual exclusion. Such a feat cannot be accomplished in Separation Logic, as far as we are aware, since it lacks the full permission.

In the Rely-Guarantee approach, a thread must specify invariants which describe how it will not interfere with particular conditions required by other threads. Simultaneously a thread must specify the non-interference conditions that it requires of other threads. When a program is correct, the rely and guarantee specifications of each thread weave together to form a global proof of correctness. However, the Rely-Guarantee approach suffers because system specifications must be written in a global manner. A thread states not only its pre and post conditions, but also which invariants of other threads it promises to not invalidate. These invariants could have nothing to do with the memory that it modifies.

Calvin-R [46] is an automation of the Rely-Guarantee concept, where the rely and guarantee predicate for every thread is a conjunction of *access predicates*, describing which locks must be held when accessing shared variables. Calvin-R uses this information, along with the Lipton [72] theory of reduction, to prove method behavioral specifications. Calvin-R must assume that every method could be called concurrently, and therefore variables must always be accessed in accordance with their access predicate. Whereas in our system, a unique permission to the receiver of a method call says that the object cannot be thread-shared for the duration of that call, and therefore fields do not require protected access. Also, this work does not mention the effect that aliasing might have on the validity of access predicates, but presumably something must be done to ensure soundness.

In recent work, Vaziri et al. [93] have proposed a system to help programmers preserve the consistency of objects with a feature called *atomic sets*. In this approach, programmers specify that certain fields of an object are related, and must be modified atomically. An interprocedural static analysis then infers code locations where synchronization is required. While a promising approach, it does not allow verification of functional properties of code, such as the correct usage of object protocols.

Finally, Harris and Jones [54] introduce a mechanism for STM Haskell that ensures a data invariants will not be violated during a given execution of a program. However, this is a dynamic technique that cannot guarantee conformance for all executions.

### 3.5.2 Race Detection

There has been much work in the automated prevention of data races.

Dynamic race detectors [86, 99] check for unordered reads and writes to the same location in memory at execution time by instrumenting program code. Model-checking approaches have also been explored [56, 88]. These work by abstractly exploring possible thread interleavings in order to find ones in which there is no ordering on a read and write to the same memory location. There have also been a number of static analyses and type systems for data race prevention [24, 39, 50, 51, 81] as well, each making trade-offs in the number of false-positives and the complexity of annotations required.

The fundamental difference between each of these race detection approaches and our approach is the presence or absence of behavioral specifications. None of the other approaches require behavioral specifications, and therefore can check only an implicit specification; that the program should contain no data races. In our system, typestate specifications, which describe the intended program behavior, allows us to prevent more semantically meaningful race conditions.

Atomicity checkers [44, 61, 85] help programmers achieve atomicity using locks, but can only ensure the atomicity that the programmer deems necessary. Given a specification of a piece of code that must execute as if atomic and specifications relating locks to the memory that they protect, an atomicity checker will tell the programmer whether or not locks are used correctly, according to the theory of reduction [72]. Once again, because atomicity checkers do not require behavioral specifications, they do not tell the program which sections of code must execute atomically in order to ensure program correctness.

## 3.6 Future Work

In the future we would like to determine what sorts of access permissions might be more useful in a thread-shared context. At the moment, permissions that are thread-shared, and permissions that are merely aliased locally are not distinguishable, and we would like to tease them apart. For instance, we would like to have a thread-local version of the share permission that would not require synchronization.

Additionally, some programmers desire more flexibility in their locking methodologies. We would like to provide them with the ability to associate particular pieces of state with particular locks, rather than the one-size-fits-all locking discipline we currently enforce. Ideally, this would be a relatively simple process, using a specification system from an existing approach [43, 50, 64]. Still, some modification to the verification rules would be required.

## 3.7 Conclusion

In this chapter we presented our approach. We have developed a programming language whose type system prevents misuse of protocols in concurrent programs. Here misuse means both improper client-side usage and inconsistent implementation. Protocols can be inadvertently used incorrectly in concurrent programs because of improper synchronization. For example, a client of a queue might forget to lock the queue in between the time that the queue was checked for

open-ness and the time when an item is dequeued. Given a description of the protocols defined by each class, our type system can help enforce correct synchronization.

Our overall approach is based on Bierhoff and Aldrich [15]'s access permissions methodology, which encodes abstract state and a succinct description of aliasing into the type of each program reference. We have reinterpreted the aliasing description as a thread-sharing description, which gives us a sound analysis with no additional specification burden. As part of our approach, programs must adhere to a simple but common locking protocol.

In this chapter we formalized our analysis as a type system for a simple object-oriented language. In the next chapter we will take a simplified version of this language and prove that it is type-safe. The result of this proof is a guarantee: if no type errors are issued, we can promise that the defined protocols are being obeyed, even if the face of concurrent access.

Finally, we described a static analysis for Java programs based on our approach, Sync-or-Swim. In subsequent chapters we will evaluate our approach by using this tool to verify a number of open-source programs.

# Chapter 4

# Proof of Soundness

*Everybody loves progress but nobody likes change.*

---

## 4.1 Summary

This chapter contains a proof of soundness for the language presented in the proceeding chapter. This proof is for a language that uses synchronized blocks as the means of mutual exclusion. In previous work [9] we presented a proof of type safety for a similar language using atomic blocks as the means of mutual exclusion.

As in other proofs of type safety, this proof depends on two interlocking pieces. The first, a proof of Progress, shows that any well-typed program that has not completed can continue running by taking a single step. The second piece, a proof of Preservation, says that any program that can take a step will take a step to another well-typed program. By putting the two pieces together, we prove to ourselves that any program written in our language that satisfies the typing rules will run to completion without getting "stuck" in some poorly defined state. But as part of this process, the proof formalizes many of the invariants that we have described in the previous chapter. Namely, that any time a thread has a permission certifying that an object in memory is in a certain abstract state, that object is *actually* in that state in memory, even if shared amongst multiple, modifying threads. These facts, here expressed as formal relationships between the run-time state and the contexts in which threads are typed, are primarily what is interesting about the proof.

This section contains a summary of the proof for those readers who are not interested in the complete technical details. The general features of the proof are fairly straightforward. It is a proof of type safety for a language with a small-step, structural operational semantics. The language that we prove sound is a simplified version of the language described in the proceeding and subsequent chapters. This was mostly done to reduce the amount of effort involved in the proof; even as it stands the proof is quite long. Our goal was to strike a balance, to simplify the language to the point where nothing is extraneous but to retain enough of the core features of language so that the proof retains its explanatory power.

The language presented here has only three permissions, rather than five. But the permissions it has, unique, share, and immutable are the most interesting with respect to thread sharing. Fractions have been removed, as they have previously been proven sound in another quite similar language [15]. Dimensions, state hierarchies and state guarantees have also been forgone, and the language of specification has been greatly simplified.

This chapter contains a formal operational semantics for the proof language, and it contains one interesting feature. Each object in memory, in addition to holding the values of its fields, keeps track of its current abstract state, or if it is unpacked. In an actual implementation of the language, this information is unnecessary at run-time. However, we need them for the main result of our proof; to show that an object will be in a certain state when an access permission says statically that it will.

The proof itself is structured into two levels for ease of understanding. At the top level of the proof, the entire pool of threads that constitutes a running program takes steps from one program state to the next. At this level, the proof of safety guarantees a number of global properties, for example that any given lock is held by at most one thread. The top level proof depends critically on the proof of safety for a single thread, to which it defers when a single thread within the thread pool takes a step.

The proof of safety for a single thread is structured in a rely/guarantee fashion. Through the statement of single-threaded preservation (Theorem 4) we show that a thread can only take a certain number of actions, in particular ones that will not interfere with the actions of other threads. In return, it can "rely" on (is given) certain facts in the global state not changing. This allows the proof of single-threaded safety to proceed without the need to worry how each action will affect other threads in the program. These guarantees come to fruition at the top level of the proof, as the preservation theorem (Theorem 2) depends on one thread's step not affecting the well-typedness of the other threads that did not take a step.

The invariants maintained by a thread in the single-threaded proof of preservation are really the substance of the proof. After each step, we are forced to show that there are typing contexts in which the new thread expression can be well-typed. Moreover, we must prove a heap invariant, which says that these new typing contexts are consistent with the actual run-time state of the program. It is the truth of the heap invariant that makes our language interesting at all. The most important aspect mandated by the heap invariant is that every permission inside a thread's typing context must be accurate with respect to the heap. If the permission says that an object is in the "Open" abstract state, than this must be true of the object in the heap. In order to prove this invariant, other invariants are needed. For example, an invariant that says each lock is held by at most one thread. Another invariant tells us that a thread can have specific state information about a share permission only if it holds a lock for that object. Same goes for the unpacking of a share permission. And finally, in order to provide guarantees for other threads, we prove that a thread will not modify the states of objects to which it has no permission, or immutable permission. By weaving these facts together, our proof ensures that any thread-shared object is always in the state specified by its type.

As previously mentioned, the language used in our proof does not include full and pure permissions, nor fractions, nor states hierarchies, dimensions and guarantees. While our proof does not include these features, we believe that the language is sound with them added. If we were to extend the proof to include these features, a great deal more proof machinery would become nec-

essary, but the basics would not change. In order to allow full and pure permission to an object, we would need a heap invariant to ensure that references associated with full and purepermission statically were not unpacked unless dynamically a lock was held by the associated thread. Additionally, we would need an invariant saying that no more than one pure permission could track the state of an object at a time, and that thread must hold the lock for the object. This is identical to our current restriction for share permissions. Fractions have been proven sound in a very similar context [15]. We would need an invariant to ensure that all of the available fractions to an object never exceeded the value 1. A similar change would be required for dimensions. Our invariants would be changed to allow permissions exceeding 1 to the same object as long as those permissions were to different dimensions. State hierarchies otherwise would pose little trouble.

One final point of interest is something that our proof does not say. We do not prove the absence of deadlocks. In our language, a thread can still take a step even if it wants to grab a lock that is held by another thread. The step is simply a transition back to its previous configuration. While deadlocks are a common problem with shared-memory concurrency, we do not address them in this work, and it is important to keep in mind that just because type safety says a program can always take a step does not ensure that it will make progress.

## 4.2 Language Definition

In this section we define the syntax and static and dynamic semantics for the proof language. Section 4.3 contains the proof of type safety. While the language itself is similar to the language presented in the proceeding chapter, it has a number of differences which we describe in the next section.

### 4.2.1 Language Differences and Simplifications

As mentioned at the beginning of this chapter, the proof language is a simplified version of the language presented in the previous chapter. It lacks interesting state hierarchies, so state dimensions and guarantees cannot be used. Instead of a top-most alive state, there is an unknown state, ?. Specifications are also greatly simplified. Rather than general lineal logic specifications, a much simpler language is permitted that effectively permits just linear conjunction. This language also lacks fractions, so permissions cannot be reassembled.

There are other differences as well. This language contains no conditional ('if') expression. While conditionals are necessary for just about any interesting program, they do not add much of interest to the proof and would require us to add Boolean types to the language. In the proof language, certain expressions (specifically, field reads) are annotated with the amount of permission that they use. This makes the proof slightly simpler and does not end up being much of a restriction since an implementation could add these annotations automatically (much as Plural currently does). In the same vein, this language has three different versions of the `unpack` and `inunpack` expressions, one for each permission kind. This makes the proof slightly simpler in a few places. Note also that in the proof language any reference can be unpacked, while in the language previously presented only the receiver (i.e., `this`) could be unpacked.

Finally, like many proofs of type safety for languages with references, our language has a notion of an object label, $o$. As in other systems, object labels in our language help ensure that programs described in our proof are closed even when they depend on heap cells to be well-typed. However, because of permissions, our language has an additional layer of indirection. In our language, a closed expression can contain indirect references $l$. These indirect references can be associated with access permissions in the linear context, much as variables can. A run-time context, $\rho$, keeps track of the mapping between indirect references $l$ and object labels $o$. When multiple indirect references point to the same object label, this indicates that the object is aliased. The multiple indirect references allow us to keep track of different permission to the same object. This approach was borrowed from recent work [47].

### 4.2.2 Syntax

Figure 4.1 presents the syntax for our proof language. Programs are the same, but class definitions are simplified. Because there are no state hierarchies, fields no longer need to be declared as being in some state, so their declaration, $F$, is quite simple. Constructors, $I$, have been simplified. One now simply declares the state that the object will be in upon instantiation. Classes no longer allow the declaration of dimensions and refining states. State declaration and state invariant declaration have been rolled into one syntactic category, $N$. A state is declared, and then associated with a list of fields and associated permissions, $p$. The idea is that this list determines the permissions that must be available to the associated field when the object is in the declared state.

Method declarations, $M$, look quite different. Our language no longer allows general linear logic specifications, meaning that the interesting connectives have been removed. This greatly simplifies the proof because now the statements that can be made about permissions in the linear context can be largely syntactic. (We only have to worry about whether or not the context contains a certain permission, without having to dive down into each predicate to see if it contains that permission conjoined with some other permission.) This change is reflected in the method specification. Each parameter is annotated with a specification, $E \gg p$. $E$, as before, is an expression type, which includes both a class type and a permission, in this case the permission that must hold for the parameter before the method call. The second permission is the permission that will be true for that parameter after the method returns. Pre- and post-condition permissions for the method receiver are written inside of the braces, $[p_t \gg p'_t]$. The return type, $E_r$, is an expression type because it includes both the class type for the returned object as well as the permission that is returned for that object.

The Boolean type has been removed from our language and so the term category, $t$, now just includes variables $x$ and indirect references $l$. The idea of indirect references was borrowed from an earlier proof for a permission-based language [47]. It allows us to keep track of the multiple permissions that a thread may have to the same object. We cannot just associate permissions with variables because they will be substituted away at run-time. Yet we still must track how many permissions a thread has to the actual object label $o$. In our earlier proof of soundness [9], we used an abstraction of a stack for dealing with this issue, and it was much less elegant. Object labels, $o$, are the only values in our language.

Expressions, $e$, are largely the same as in the previous chapter. Field reads, $t.f^k$ are now

$$
\begin{array}{rrcl}
\textit{program} & PG & ::= & \langle \overline{CL}, e \rangle \\
\textit{class decls.} & CL & ::= & \texttt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\} \\
\textit{field decls.} & F & ::= & f : T \\
\textit{initial state} & I & ::= & \texttt{initially}\ \langle s \rangle \\
\textit{state inv.} & N & ::= & s = \overline{f : p} \\
\textit{methods} & M & ::= & E_r\ m(\overline{E \gg p\ x})[p_t \gg p'_t] = e \\
\textit{expr types} & E & ::= & C.p \\
\textit{terms} & t & ::= & x \mid l \\
\textit{values} & v & ::= & o \\
\textit{Ctx. Bindings} & b & ::= & t \mid o \\
\textit{expressions} & e & ::= & t \mid o \mid t.f^k \mid t.f := t \\
 & & \mid & \texttt{new}\ C(\overline{t}) \mid t.m(\overline{t}) \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid & \texttt{spawn}\ (t.m(\overline{t})) \mid \texttt{synchronized}\ (t)\ e \mid \texttt{insync}(l)\ e \\
 & & \mid & \texttt{unpackuniq}(t, S, s)\ \texttt{in}\ e \mid \texttt{inunpackuniq}(l; S; s)\ e \\
 & & \mid & \texttt{unpackshare}(t, S, s)\ \texttt{in}\ e \mid \texttt{inunpackshare}(l; S; s)\ e \\
 & & \mid & \texttt{unpackimm}(t, S)\ \texttt{in}\ e \mid \texttt{inunpackimm}(l; s)\ e \\
\textit{lock contexts} & \Psi & ::= & \bullet \mid \Psi, t \\
\textit{valid contexts} & \Gamma & ::= & \bullet \mid \Gamma, b : C \\
\textit{linear contexts} & \Delta & ::= & \bullet \mid \Delta, b : p \\
\textit{packing state} & u & ::= & \mathsf{up}(t; k; S; \overline{f : p}) \mid \mathsf{p} \\
\textit{heaps} & H & ::= & \bullet \mid H, o \mapsto C(\overline{o})@\$ \\
\textit{heap flags} & \$ & ::= & s \mid \mathsf{up} \mid \mathsf{ro}(s) \\
\textit{run-time locks} & \kappa & ::= & \bullet \mid \kappa, o \overset{i}{\mapsto} \iota \\
\textit{label map} & \rho & ::= & \bullet \mid \rho, l \mapsto o \\
\textit{thread pools} & L & ::= & \bullet \mid L, \iota.e
\end{array}
$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *classes* | $C$ | *fields* | $f$ | *variables* | $x, y, z$ | *objects* | $o$ |
| *methods* | $m$ | *states* | $s$ | *indirect refs.* | $l$ | *thread ids* | $\iota$ |

Figure 4.1: The syntax of our proof language. $p$, $k$, and $S$ appear in Figure 4.2.

annotated with the amount of permission that they read. This simplifies the typing rules somewhat and can easily be put in automatically in an implementation. There are now three different unpack statements, one for each permission kind. This simplifies the proof by making it immediately clear from the syntax what kind of permission was used to unpack the object. There are also new syntactic forms, `insync`, `inunpackuniq`, `inunpackshare`, and `inunpackimm`. Threads will transition to these expressions after acquiring a lock or unpacking an object, as the case may be. By syntactically differentiating the threads that are currently executing with locks or with unpacked objects, certain invariants about program structure are easier to maintain.

Locking contexts, valid contexts and linear contexts are the same, except that they can now in certain cases reference $o$ and $l$. The locking context can never contain object labels, however. We assume implicitly that static contexts $\Delta$ and $\Gamma$ cannot have duplicate entries. $\Psi$, on the other hand, may have duplicate entries if the same lock is acquired multiple times. Packing contexts

have been enhanced because the linear context can no longer hold field permissions. This change made various aspects of the proof more syntax-driven (a repeating theme in our simplifications). A packing context, $u$, can be packed, p, or it can be unpacked. When unpacked, it contains the unpacked term, $t$, the fraction with which it was unpacked, the state from which it was unpacked, and a list of fields of the unpacked object and the remaining permission available to each one.

The remaining syntactic elements are actually run-time data structures. Heaps, $H$, are a map from object labels to object values, where an object value holds a class, a list of object labels corresponding to the current fields of the object, and a heap flag. Heap flags, \$, are our way of tracking dynamically the state of each object. An object may be in a state $s$, or it may be unpacked, up, or unpacked in a read-only state, ro($s$). Because our language contains no fractions, and permissions cannot be rejoined, once an object is unpacked with an immutable permission, it will remain unpacked for the rest of the program lifetime. The run-time locking context $\kappa$ keeps track of which locks are held by which threads by mapping object labels to thread identifiers, $\iota$. If an object is not in the domain of $\kappa$, then it is not held by any thread. (And as in Java, each object in the heap is implicitly a lock.) A lock counter $i$ keeps track of how many times a thread has acquired the same lock at a given moment. This number is implicitly greater than zero.

$\rho$ is a run-time mapping between indirect object references $l$, and object labels $o$. And finally, a thread pool, $L$, is a list of threads where each thread is an expression and each is given a thread identifier, $\iota$.

### 4.2.3 Permission Syntax

$$
\begin{array}{rrcl}
\textit{permissions} & p & ::= & k@S \\
\textit{states} & S & ::= & s \mid ? \\
 & k & ::= & \mathsf{unique} \mid \mathsf{immutable} \mid \mathsf{share}
\end{array}
$$

Figure 4.2: Permission syntax for the proof language

As mentioned, permissions are greatly simplified in the proof language because of the lack of dimensions, fractions and guarantees. Permissions in the proof language no longer mention the reference with which they are associated. Instead, permissions are associated with references inside the linear context much like traditional types, $\Delta, t : p$. This simplification can be done because there are no interesting connectives in the specification language. A permission, $p$, is nothing more than a permission kind and a state. States, $S$, can be either a named state or the special state ?, the unknown state. This state is always less precise than an actual state. During the process of forgetting, share permissions will be downgraded to this state. Finally, there are three permission kinds, unique, immutable and share.

### 4.2.4 Permission Well-Formedness and Manipulation

As before, there are a number of rules that allow permissions to be manipulated in various ways. In general, they are simpler in the proof language than in the full language.

**Permission Well-Formedness** The judgment for ensuring permission well-formedness is greatly simplified due to the removal of fractions and state hierarchies. Now, for a permission to be well-formed, it is sufficient to show that the state that it mentions is either the unknown state, $?$, or a state that is actually defined for the given reference type.

$$\boxed{\Gamma \vdash \quad t : p \ \mathsf{wf}}$$

$$\mathsf{states}(C) \quad = \quad \{s_i \mid s_i = \overline{f : p} \in \overline{N}\} \quad \text{where } \texttt{class}\ C\{\ldots \overline{N} \ldots\}$$

$$\Gamma \vdash \quad t : k@? \ \mathsf{wf} \qquad\qquad \frac{s \in \mathsf{states}(C)}{\Gamma, t : C \vdash \quad t : k@s \ \mathsf{wf}}$$

**Splitting and Forgetting** The proof language contains simplified rules for splitting and forgetting. The splitting judgment says that a unique permission can be divided into either a share permission or an immutable permission, and that share and immutable permissions can be duplicated as many times as necessary. Once again, there are no rules for joining.

$$\boxed{k \Rrightarrow k/k}$$

$$\frac{k = \mathsf{share} \vee k = \mathsf{immutable}}{\mathsf{unique} \Rrightarrow k/k} \qquad\qquad \frac{k = \mathsf{share} \vee k = \mathsf{immutable}}{k \Rrightarrow k/k}$$

Of particular interest are the "forgetting" judgments, which weaken share permissions and linear contexts that contain them if they are not protected by locks. This judgment is used in-between expressions when other threads may have modified thread-shared mutable objects.

$$\boxed{\downarrow^{\Psi}(\Delta) = \Delta}$$

$$\downarrow^{\Psi}(\bullet) = \bullet \qquad \frac{t \in \Psi \qquad \downarrow^{\Psi}(\Delta) = \Delta'}{\downarrow^{\Psi}(\Delta, t{:}p) = \Delta', t{:}p} \qquad \frac{t \notin \Psi \qquad \downarrow^{\Psi}(\Delta) = \Delta' \qquad p' = \downarrow(p)}{\downarrow^{\Psi}(\Delta, t{:}p) = \Delta', t{:}p'}$$

$$\boxed{\downarrow(p) = p}$$

$$\downarrow(\mathsf{unique}@S) = \mathsf{unique}@S \qquad\qquad \downarrow(\mathsf{immutable}@S) = \mathsf{immutable}@S$$

$$\downarrow(\mathsf{share}@S) = \mathsf{share}@?$$

**Proving Permissions** Instead of a general linear logic proof judgment, this language contains a few simple rules for proving permissions. The permission kind proof judgment says that a permission kind can always be used to prove the same permission kind, or any permission kind that it can be split into.

$$\boxed{k \vdash \ k'}$$

$$k \vdash \ k \qquad\qquad \frac{k \Rrightarrow k'/k''}{k \vdash \ k'}$$

States can prove other states if they are the same state, or if the state to be proved is the unknown state $?$.

$$\boxed{S \vdash \ S'}$$

$$s \vdash \ s \qquad\qquad S \vdash \ ?$$

A permission proof simply degrades into proofs of its constituent pieces.

$$\boxed{p \vdash \ p'}$$

$$\frac{k \vdash \ k' \qquad S \vdash \ S}{k@S \vdash \ k'@S'}$$

Finally, because fields are a part of a different syntactic category, and because we will often need to prove multiple field permissions at once when packing or unpacking an object, we have a separate judgment for proving a list of field permissions from another list of field permissions. This judgment says that a list of field permissions can be proved from another list of field permission if the latter list of permissions contains every field in the former, and the associated permission is at least as strong in every case.

$$\boxed{\overline{f : p} \vdash \ \overline{f' : p'}}$$

$$\frac{\forall f'_i : p'_i \in \overline{f' : p'}, \exists f_i : p_i \in \overline{f : p} \ \text{ s.t } \ f_i = f'_i \wedge p_i \vdash \ p'_i}{\overline{f : p} \vdash \ \overline{f' : p'}}$$

**Permission Consistency** An important part of the single thread and thread pool heap invariants is ensuring consistency between permission facts. This notion of consistency is related to, but different than permission proving. Consistent permissions are simply permissions that can coexist because they do not directly contradict one another. Such a notion of consistency exists for permission states:

$$\boxed{S \leftrightarrow S'}$$

$$\frac{S \vdash \ S'}{S \leftrightarrow S'} \qquad\qquad \frac{S' \vdash \ S}{S \leftrightarrow S'}$$

And consistency exists for permission kinds:

$$\boxed{k \leftrightarrow k'}$$

$$\mathsf{immutable} \leftrightarrow \mathsf{immutable} \qquad\qquad \mathsf{share} \leftrightarrow \mathsf{share}$$

Note that for permission kinds, multiple unique permissions cannot be consistent. These notions are used both in the later heap invariant definitions as well as the rules defining dynamic semantics.

### 4.2.5 Type-Checking and Program Well-Formedness

**Helper Judgments for Type-Checking** As before, when an immutable object is unpacked, we must unpack its fields in a read-only state. This is done via the purify function.

$$\boxed{\mathsf{purify}(\overline{f : p}) = \overline{f : p}}$$

$$\begin{array}{lcl}
\mathsf{purify}(\overline{f : p}, f_n : \mathsf{share}@S) & = & \mathsf{purify}(\overline{f : p}) \\
\mathsf{purify}(\overline{f : p}, f_n : \mathsf{unique}@S) & = & \mathsf{purify}(\overline{f : p}), f_n : \mathsf{immutable}@S \\
\mathsf{purify}(\overline{f : p}, f_n : \mathsf{immutable}@S) & = & \mathsf{purify}(\overline{f : p}), f_n : \mathsf{immutable}@S \\
\mathsf{purify}(\bullet) & = & \bullet
\end{array}$$

We also need a function for looking up the state invariants for object states. As before this function is called, $\mathsf{inv}_C$. Interestingly, it can be used to look up an invariant for the unknown state ?, but the result is an empty predicate. Such an ability is necessary to code dynamic state test methods like `is_closed` in the previous chapter, and to do so even in our proof language.

$$\boxed{\mathsf{inv}_C(S) = \overline{f : p}}$$

$$\mathsf{inv}_C(?) = \bullet \qquad \frac{\texttt{class}\ C\ \{\overline{F}\ I\ \overline{N}\ \overline{M}\} \qquad s = \overline{f : k@S} \in \overline{N}}{\mathsf{inv}_C(s) = \overline{f : k@S}}$$

At various points in our proof, specific subexpressions contained in an expression will be important. In particular, those expressions that only exist in running programs and cannot be written by programmers, `insync`, `inunpackuniq` and its ilk, can only appear in certain configurations and must correspond in a very direct way with the static typing contexts. For this reason we will use two functions, activeLocks and activeUnpack, in both the typing rules and in our theorems of type-safety. The functions return the objects that an expression has currently locked or unpacked, respectively. In addition, they set rules for expressions being well-defined. For example, the definition of activeLocks(`synchronized` $(t)\ e$) requires that there be no active locks in $e$.

activeLocks($e$)

| | | | |
|---|---|---|---|
| activeLocks(`insync`$(l)\ e$) | $=$ | $\{l\} \cup$ activeLocks$(e)$ | |
| activeLocks$(t)$ | $=$ | $\emptyset$ | |
| activeLocks$(t.f^k)$ | $=$ | $\emptyset$ | |
| activeLocks$(t.f := t')$ | $=$ | $\emptyset$ | |
| activeLocks(`new` $C(\bar{t})$) | $=$ | $\emptyset$ | |
| activeLocks$(t.m(\bar{t}))$ | $=$ | $\emptyset$ | |
| activeLocks(`let` $x = e_1$ `in` $e_2$) | $=$ | activeLocks$(e_1)$ | |
| activeLocks(`spawn`$(t.m(\bar{t}))$) | $=$ | $\emptyset$ | |
| activeLocks(`synchronized` $(t)\ e$) | $=$ | $\emptyset$ | requires that activeLocks$(e) = \emptyset$ |
| activeLocks(`unpackuniq`$(t, S, s)$ `in` $e$) | $=$ | $\emptyset$ | requires that activeLocks$(e) = \emptyset$ |
| activeLocks(`inunpackuniq`$(l; S; s)\ e$) | $=$ | activeLocks$(e)$ | |
| activeLocks(`unpackshare`$(t, S, s)$ `in` $e$) | $=$ | $\emptyset$ | requires that activeLocks$(e) = \emptyset$ |
| activeLocks(`inunpackshare`$(l; S; s)\ e$) | $=$ | activeLocks$(e)$ | |
| activeLocks(`unpackimm`$(t, S)$ `in` $e$) | $=$ | $\emptyset$ | requires that activeLocks$(e) = \emptyset$ |
| activeLocks(`inunpackimm`$(l; s)\ e$) | $=$ | activeLocks$(e)$ | |

activeLocks$(e, \rho)$

$$\mathsf{activeLocks}(e, \rho) \quad = \quad \{o \mid o = \rho(l), l \in \mathsf{activeLocks}(e)\}$$

activeUnpack$(e, k)$

$$
\begin{array}{lcl}
\mathsf{activeUnpack}(\texttt{inunpackshare}(l;S;s)\ e, \mathsf{share}) & = & \{l\} \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackshare}(l;S;s)\ e, \mathsf{immutable}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackshare}(l;S;s)\ e, \mathsf{unique}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackimm}(l;S;s)\ e, \mathsf{immutable}) & = & \{l\} \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackimm}(l;S;s)\ e, \mathsf{share}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackimm}(l;S;s)\ e, \mathsf{unique}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackuniq}(l;S;s)\ e, \mathsf{unique}) & = & \{l\} \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackuniq}(l;S;s)\ e, \mathsf{immutable}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{inunpackuniq}(l;S;s)\ e, \mathsf{share}) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, k) & = & \mathsf{activeUnpack}(e_1, k) \\
\mathsf{activeUnpack}(\texttt{insync}(l)\ e, k) & = & \mathsf{activeUnpack}(e, k) \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{unpackshare}(t,S,s)\ \texttt{in}\ e, k) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(\texttt{unpackimm}(t,S)\ \texttt{in}\ e, k) & = & \emptyset \\
 & & \text{requires that } \mathsf{activeUnpack}(e) = \emptyset \\
\mathsf{activeUnpack}(e) & = & \emptyset \\
 & & \text{all other cases}
\end{array}
$$

$$\boxed{\mathsf{activeUnpack}(e, \rho, k)}$$

$$\mathsf{activeUnpack}(e, \rho, k) = \{o \mid o = \rho(l), l \in \mathsf{activeUnpack}(e, k)\}$$

$$\boxed{\mathsf{activeUnpack}(L, \rho, k)}$$

$$\mathsf{activeUnpack}(L, \rho, k) = \bigcup_{e_i \in L} \mathsf{activeUnpack}(e_i, \rho, k)$$

$$\boxed{\mathsf{activeUnpack}(e)}$$

$$\mathsf{activeUnpack}(e) = \bigcup_{k \in \{\mathsf{unique},\mathsf{full},\mathsf{immutable},\mathsf{share},\mathsf{pure}\}} \mathsf{activeUnpack}(e, k)$$

**Expression Type-Checking** The typing judgment for a single expression is quite similar to the same judgment in the full language.

$$\boxed{\Gamma; \Delta; \Psi; u \vdash \quad e : E \quad \dashv \Delta'; u'}$$

Still, for simplicity's sake, the judgment has been changed somewhat. Note the two output contexts on the right-hand side of the expression, $\Delta'$ and $u'$. $\Delta'$ contains all of the permissions that are in $\Delta$ that are not consumed by the expression $e$. Unlike in the previous chapter, where the linear context is divided in a seemingly arbitrary manner in certain rules (e.g., the rule for typing `let`), in this language all of the permissions are "threaded" through each expression. While the arbitrary dividing of permission contexts makes for a simpler presentation, it makes it somewhat more difficult to make definitive statements about the current state of the linear context.

The decision to have an output packing context was based on similar reasoning. In the proof language, if an object is unpacked then the packing context contains all of the permissions to fields of that object. The output packing context $u'$ thus contains all of the permissions to fields that are available after the current expression has been typed. (If an object is indeed unpacked, then the output unpacking context will contain some redundant information, such as the name of the unpacked variable and the permission with which it was unpacked.

Note that there is no separate term type-checking judgment, since terms are nothing but variables in the proof language.

P-TERM-I

$$\frac{k \Rrightarrow k'/k'' \qquad S \vdash \quad S'}{\Gamma, b : C; \Delta, b : k@S; \Psi; u \vdash \quad b : C.k'@S' \quad \dashv \Delta, b : k''@S; u}$$

P-TERM-II

$$\frac{S \vdash \quad S'}{\Gamma, b : C; \Delta, b : k@S; \Psi; u \vdash \quad b : C.k@S' \quad \dashv \Delta; u}$$

P-LOAD

$$\frac{\begin{array}{c} u = \mathsf{up}(t_r; k_r; S_r; \overline{f : p}, f_i : k_i@S_i) \\ \mathsf{localFields}(C, f_i) = C_i \qquad k_i \Rrightarrow k/k_i' \qquad u' = \mathsf{up}(t_r; k_r; S_r; \overline{f : p}, f_i : k_i'@S_i) \end{array}}{\Gamma, t_r{:}C; \Delta; \Psi; u \vdash \quad t_r.f_i^k : C_i : k@S_i \quad \dashv \Delta; u'}$$

P-ASSIGN

$$\frac{\begin{array}{c} u = \mathsf{up}(t_f; k_r; S_r; \overline{f : p}, f_i : k_i@S_i) \qquad \mathsf{writes}(k_r) \qquad \mathsf{localFields}(C, f_i) = C_i \\ k_i \vdash \quad k \qquad \Gamma, t_r{:}C; \Delta; \Psi; u \vdash \quad t : C_i.p \quad \dashv \Delta'; u \qquad u' = \mathsf{up}(t_f; k_r; S_r; \overline{f : p}, f_i :\downarrow (p)) \end{array}}{\Gamma, t_r{:}C; \Delta; \Psi; u \vdash \quad t_r.f_i := t : C_i.k@S_i \quad \dashv \Delta'; u'}$$

P-UNPACK-UNIQUE

$$\frac{\begin{array}{c} \Gamma, t; C; \Delta; \Psi; \mathsf{up}(t, \mathsf{unique}, S, \mathsf{inv}_C(S)) \vdash \quad e : E \quad \dashv \Delta'; \mathsf{up}(t, \mathsf{unique}, S, \overline{f' : p'}) \\ \overline{f' : p'} \vdash \quad \mathsf{inv}_C(s) \end{array}}{\Gamma, t : C; \Delta, t : \mathsf{unique}@S; \Psi; \mathsf{p} \vdash \quad \texttt{unpackuniq}(t, S, s)\ \texttt{in}\ e : E \quad \dashv \Delta', t{:}\mathsf{unique}@s; \mathsf{p}}$$

P-INUNPACK-UNIQ

$$\frac{\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l, \mathsf{unique}, S, \overline{f : p}) \vdash \quad e : E \quad \dashv \Delta'; \mathsf{up}(l, \mathsf{unique}, S, \overline{f' : p'}) \qquad \overline{f' : p'} \vdash \quad \mathsf{inv}_C(s)}{\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l, \mathsf{unique}, S, \overline{f : p}) \vdash \quad \texttt{inunpackuniq}(l; S; s)\ e : E \quad \dashv \Delta', l{:}\mathsf{unique}@s; \mathsf{p}}$$

P-UNPACK-IMM

$$\frac{p_r = k_r@s \qquad k_r \Rrightarrow \mathsf{immutable}/k' \qquad \Gamma, t_r : C_r; \Delta; \Psi; \mathsf{up}(t_r, \mathsf{immutable}, s, \mathsf{purify}(\mathsf{inv}_{C_r}(s))) \vdash \ e : E \ \dashv \Delta'; \mathsf{up}(t_r, \mathsf{immutable}, s, \overline{f' : p'}) \qquad \overline{f' : p'} \vdash \ \mathsf{purify}(\mathsf{inv}_{C_r}(s))}{\Gamma, t_r : C_r; \Delta, t_r : p_r; \Psi; \mathsf{p} \vdash \ \texttt{unpackimm}(t_r, s) \ \texttt{in} \ e : E \ \dashv \Delta', t_r{:}\mathsf{immutable}@s; \mathsf{p}}$$

P-INUNPACK-IMM

$$\frac{\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l, \mathsf{immutable}, s, \overline{f : p})) \vdash \ e : E \ \dashv \Delta'; \mathsf{up}(l, \mathsf{immutable}, s, \overline{f' : p'}) \qquad \overline{f' : p'} \vdash \ \mathsf{purify}(\mathsf{inv}_C(s))}{\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l, \mathsf{immutable}, s, \overline{f : p}) \vdash \ \texttt{inunpackimm}(l; s) \ e : E \ \dashv \Delta', l{:}\mathsf{immutable}@s; \mathsf{p}}$$

P-UNPACK-SHARE

$$\frac{p_r = k_r@S \qquad k_r \Rrightarrow \mathsf{share}/k' \qquad \Gamma, t_r : C_r; \downarrow^{\bullet} (\Delta); \Psi, t_r; \mathsf{up}(t_r, \mathsf{share}, S, \mathsf{inv}_C(S)) \vdash \ e : E \ \dashv \Delta'; \mathsf{up}(t_r, \mathsf{share}, S, \overline{f' : p'}) \qquad \overline{f' : p'} \vdash \ \mathsf{inv}_{C_r}(s')}{\Gamma, t_r : C_r; \Delta, t_r : p_r; \Psi, t_r; \mathsf{p} \vdash \ \texttt{unpackshare}(t_r, S, s') \ \texttt{in} \ e : E \ \dashv \Delta', t_r{:}\mathsf{share}@s'; \mathsf{p}}$$

P-INUNPACK-SHARE

$$\frac{\Gamma, l{:}C; \Delta; \Psi, l; \mathsf{up}(l, \mathsf{share}, S, \overline{f : p}) \vdash \ e : E \ \dashv \Delta'; \mathsf{up}(l, \mathsf{share}, S, \overline{f' : p'}) \qquad \overline{f' : p'} \vdash \ \mathsf{inv}_C(s)}{\Gamma, l{:}C; \Delta; \Psi, l; \mathsf{up}(l, \mathsf{share}, S, \overline{f : p}) \vdash \ \texttt{inunpackshare}(l; S, s) \ e : E \ \dashv \Delta', l{:}\mathsf{share}@s; \mathsf{p}}$$

P-SYNC

$$\frac{\Gamma, t : C; \Delta; \Psi, t; u \vdash \ e : C'.p \ \dashv \Delta'; u'}{\Gamma, t : C; \Delta; \Psi; u \vdash \ \texttt{synchronized} \ (t) \ e : C'.p \ \dashv \Delta'; u'}$$

P-INSYNC

$$\frac{\Gamma; \Delta; \Psi, l; u \vdash \ e : C.p \ \dashv \Delta'; u'}{\Gamma; \Delta; \Psi, l; u \vdash \ \texttt{insync}(l) \ e : C.p \ \dashv \Delta'; u'}$$

P-LET

$$\frac{\Psi = \mathsf{activeLocks}(e_1), \Psi_2 \qquad \Gamma; \Delta; \Psi; u \vdash \ e_1 : C_1.p_1 \ \dashv \Delta'; u' \qquad \Gamma, x : C_1; \downarrow^{\Psi_2} (\Delta'), x :\downarrow (p_1); \Psi_2; u' \vdash \ e_2 : C_2.p_2 \ \dashv \Delta'', x : p_x; u''}{\Gamma; \Delta; \Psi; u \vdash \ \texttt{let} \ x = e_2 \ \texttt{in} \ e_2 : C_2.p_2 \ \dashv \Delta''; u''}$$

P-CALL

$$\frac{\Delta; \Gamma; \Psi; \mathsf{p} \vdash \ t : C.p, \overline{t : C.p} \ \dashv \Delta'; \mathsf{p} \qquad \mathsf{mtype}(m, C) = C_r.p_r \ m(\overline{C.p \gg p' \ x})[p \gg p']}{\Gamma; \Delta; \Psi; \mathsf{p} \vdash \ t.m(\bar{t}) : C_r.p_r \ \dashv \Delta', t{:}p'\overline{t : p'}; \mathsf{p}}$$

P-SPAWN

$$\frac{\Delta; \Gamma; \Psi; \mathsf{p} \vdash \ t : E, \overline{t : E} \ \dashv \Delta'; \mathsf{p} \qquad \downarrow^{\bullet} (t : E, \overline{t : E}) = t : C.p, \overline{t : C.p} \qquad \mathsf{mtype}(m, C) = C_r.p_r \ m(\overline{C.p \gg p' \ x})[p \gg p']}{\Gamma; \Delta; \Psi; \mathsf{p} \vdash \ \texttt{spawn} \ (t.m(\bar{t})) : \mathsf{Object.unique}@? \ \dashv \Delta'; \mathsf{p}}$$

P-NEW

$$\frac{\mathsf{init}(C) = \langle \overline{f : C.p}, s \rangle \qquad \Gamma; \Delta; \Psi; u \vdash \ \overline{t : C.p} \ \dashv \Delta'; u}{\Gamma; \Delta; \Psi; u \vdash \texttt{new} \ C(\bar{t}) : C.\mathsf{unique}@s \ \dashv \Delta'; u}$$

**Thread Pool Typing** Our language has a two-level hierarchy, as inspired by the AtomsFamily languages [75]. At the top level, a large thread pool takes steps. At the bottom level, individual threads take steps and, potentially, create new threads. The static semantics of each level are related in important ways. The invariants established at the top level prove that our heap is globally consistent with respect to each of the threads. The invariants established at the thread-level ensure that when a thread takes a step it cannot possibly be violating any rules expected of other threads.

We have a top-level typing rule for the thread pool.

$$\boxed{\gamma; \delta; \psi; U \vdash \quad L : \epsilon \quad \dashv \delta_o; U_o}$$

Each static context at the top level is simply a mapping from a thread identifier, $\iota$, to a more familiar context. Specifically:

$$\begin{array}{rcl} \gamma & : & \iota \times \Gamma \\ \delta & : & \iota \times \Delta \\ \psi & : & \iota \times \Psi \\ \epsilon & : & \iota \times E \\ U & : & \iota \times u \end{array}$$

The definition of the top level typing rule is rather simple. It merely delegates to the thread typing rule. (The interesting cross-thread invariants are imposed later when we discuss heap invariants.)

$$\frac{\gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \quad e : \epsilon(\iota) \quad \dashv \delta_o(\iota); U_o(\iota) \qquad \gamma; \delta; \psi; U \vdash \quad L : \epsilon \quad \dashv \delta_o; U_o}{\gamma; \delta; \psi; U \vdash \quad L, \iota.e : \epsilon \quad \dashv \delta_o; U_o}$$

$$\gamma; \delta; \psi; U \vdash \quad \bullet : \epsilon \quad \dashv \delta_o; U_o$$

**Program Well-Formedness** This next set of rules are general program well-formedness rules, and help to ensure that an entire program is well-defined.

$$\frac{\text{P-PROG} \quad \overline{CL}\ \mathsf{ok} \qquad \bullet; \bullet; \bullet; \mathsf{p} \vdash \quad e : E \quad \dashv \Delta; \mathsf{p}}{\langle \overline{CL}, e \rangle\ \mathsf{ok}} \qquad \frac{\text{P-CLASS} \quad \overline{F}\ \mathsf{ok\ in}\ C \ldots \overline{M}\ \mathsf{ok\ in}\ C}{\mathtt{class}\ C\ \{\ \overline{F}\ I\ \overline{N}\ \overline{M}\}\ \mathsf{ok}}$$

$$\frac{\text{P-FDECL} \quad f_i \text{ is unique} \qquad C_i \in \overline{CL} \cup \{\mathsf{Object}\}}{f_i : C_i\ \mathsf{ok\ in}\ C}$$

$$\text{P-CONSTR}\ \frac{\texttt{class}\ C\{\ldots s = \overline{f : p}\ldots\} \in \overline{CL}}{\texttt{initially}\langle s\rangle\ \mathsf{ok\ in}\ C}$$

$$\text{P-SINV}\ \frac{s_i\text{unique} \qquad \mathsf{localFields}(C, f_i) = C_i \qquad k_i = \mathsf{share} \Rightarrow S_i = ? \qquad f_i{:}C_i \vdash\ \ f_i{:}k_i@S_i\ \mathsf{wf}}{s_i = \overline{f : k@S}\ \mathsf{ok\ in}\ C}$$

$$\text{P-METH-DECL}\ \frac{\begin{array}{l}\mathsf{activeLocks}(e) = \emptyset \qquad \mathsf{activeUnpack}(e) = \emptyset \qquad \Gamma = \texttt{this}{:}C, \overline{x{:}C} \qquad \Gamma \vdash\ \ \overline{p}, \overline{p'}, p_t, p_t'\ \mathsf{wf} \\ \texttt{result}{:}C_r \vdash\ \ \texttt{result}{:}p_r\ \mathsf{wf} \qquad \Gamma; \texttt{this}{:}p_t, \overline{x{:}p}; \bullet; \mathsf{p} \vdash\ \ e : C_r.p_r\ \ \dashv \Delta_o, \texttt{this}{:}p_t', \overline{x{:}p'}; \mathsf{p}\end{array}}{C_r.p_r m(\overline{C.p \gg p'\ x})[p_t \gg p_t'] = e\ \mathsf{ok\ in}\ C}$$

### 4.2.6 Dynamic Semantics

The dynamic semantics for this language is partially inspired by two other systems. The lock semantics comes from Terauchi [92] while the various other pieces (again, the two-level hierarchy) come from the AtomsFamily work [75].

First we present the dynamic semantics for the thread pools. This judgment says that a thread pool $L$ in a dynamic state featuring a reference map $\rho$, a lock context $\kappa$ and a heap $H$ can take a step to new program state which contains a new reference map $\rho'$, a new locking context $\kappa'$, a new heap $H'$ and a new thread pool $L'$. The rules defining this judgment are rather simple, delegating to the expression stepping rules. Note that there are two rules, depending on whether or not the stepping thread spawns a new thread.

$$\boxed{(\rho; \kappa; H; L) \rightarrow (\rho'; \kappa'; H'; L')}$$

$$\text{D-Top-Normal}\ \frac{(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa' H'; \iota.e'; \bullet)}{(\rho; \kappa; H; L, \iota.e) \rightarrow (\rho'; \kappa'; H'; L.\iota.e')}$$

$$\text{D-TOP-SPAWN}\ \frac{\hat{\iota}\ \mathsf{fresh} \qquad (\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H'; \iota.e'; e_2)}{(\rho; \kappa; H; L, \iota.e) \rightarrow (\rho'; \kappa'; H'; L, \iota.e', \hat{\iota}.e_2)}$$

Next, the judgment for a single thread taking a step. (Note that the last field of the resulting tuple, $a$ can be either an expression, $e$, or nothing, $\bullet$, depending on whether a new thread is spawned.) This judgment says that a single thread $e$ with thread identifier $\iota$ in a program state featuring reference map $\rho$, lock context $\kappa$ and heap $H$ can take a step to a new expression $e'$ in a new dynamic context, featuring reference map $\rho'$, lock context $\kappa'$ and heap $H'$. As a result it will also produce $a$, which may or may not be a newly spawned thread.

$$\boxed{(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H'; \iota.e'; a)}$$

D-LOOKUP
$$(\rho; \kappa; H; \iota.l) \to (\rho; \kappa; H; \iota.\rho(l); \bullet)$$

D-LOAD
$$\frac{H(\rho(l)) = C(\overline{o})@\$ \qquad \mathsf{localFields}(C, f_i) = C_i}{(\rho; \kappa; H; \iota.l.f_i^k) \to (\rho; \kappa; H; \iota.o_i; \bullet)}$$

D-ASSIGN
$$\frac{H(\rho(l_1)) = C(\overline{o})@\$ \qquad \mathsf{localFields}(C, f_i) = C_i}{(\rho; \kappa; H; \iota.l_1.f_i := l_2) \to (\rho; \kappa; H[\rho(l_1) \mapsto [\rho(l_2)/o_i]C(\overline{o})]; o_i; \bullet)}$$

D-SYNC-BUSY
$$\frac{\rho(l) \mapsto \hat{\iota} \in \kappa \qquad \iota \neq \hat{\iota}}{(\rho; \kappa; H; \iota.\texttt{synchronized}\ (l)\ e) \to (\rho; \kappa; H; \iota.\texttt{synchronized}\ (l)\ e; \bullet)}$$

D-SYNC-ACQ
$$\frac{\rho(l) \notin \mathrm{dom}(\kappa)}{(\rho; \kappa; H; \iota.\texttt{synchronized}\ (l)\ e) \to (\rho; \kappa, \rho(l) \overset{1}{\mapsto} \iota; H; \iota.\texttt{insync}(l)\ e; \bullet)}$$

D-SYNC-ALREADY
$$\frac{\rho(l) \overset{i}{\mapsto} \iota \in \kappa}{(\rho; \kappa; H; \iota.\texttt{synchronized}\ (l)\ e) \to (\rho; \kappa[o \overset{i+1}{\mapsto} \iota]; H; \iota.\texttt{insync}(l)\ e; \bullet)}$$

D-INSYNC
$$\frac{(\rho; \kappa; H; \iota.e) \to (\rho'; \kappa'; H'; \iota.e'; a)}{(\rho; \kappa; H; \iota.\texttt{insync}(l)\ e) \to (\rho'; \kappa'; H'; \iota.\texttt{insync}(l)\ e'; a)}$$

D-SYNC-RELEASE-I
$$\frac{\kappa = \kappa', \rho(l) \overset{1}{\mapsto} \iota}{(\rho; \kappa; H; \iota.\texttt{insync}(l)\ o) \to (\rho; \kappa'; H; \iota.o; \bullet)}$$

D-SYNC-RELEASE-II
$$\frac{\kappa = \kappa', \rho(l) \overset{i}{\mapsto} \iota \qquad i > 1}{(\rho; \kappa; H; \iota.\texttt{insync}(l)\ o) \to (\rho; \kappa', \rho(l) \overset{i-1}{\mapsto} \iota; H; \iota.o; \bullet)}$$

D-UNPACKUNIQ-ENTER
$$\frac{H(\rho(l)) = C(\overline{o})@s_1 \qquad S \leftrightarrow s_1 \qquad H' = H[\rho(l) \mapsto C(\overline{o})@\mathsf{up}]}{(\rho; \kappa; H; \iota.\texttt{unpackuniq}(l, S, s)\ \texttt{in}\ e) \to (\rho; \kappa; H'; \iota.\texttt{inunpackuniq}(l; S; s)\ e; \bullet)}$$

D-UNPACKIMM-ENTER
$$\frac{H(\rho(l)) = C(\overline{o})@\$ \qquad \$ = s \vee \$ = \mathsf{ro}(s) \qquad H' = H[\rho(l) \mapsto C(\overline{o})@\mathsf{ro}(s)]}{(\rho; \kappa; H; \iota.\texttt{unpackimm}(l, s)\ \texttt{in}\ e) \to (\rho; \kappa; H'; \iota.\texttt{inunpackimm}(l; s)\ e; \bullet)}$$

D-UNPACKSHARE-ENTER
$$\frac{H(\rho(l)) = C(\overline{o})@s_1 \qquad S \leftrightarrow s_1 \qquad H' = H[\rho(l) \mapsto C(\overline{o})@\mathsf{up}]}{(\rho; \kappa; H; \iota.\texttt{unpackshare}(l, S, s)\ \texttt{in}\ e) \to (\rho; \kappa; H'; \iota.\texttt{inunpackshare}(l; S; s)\ e; \bullet)}$$

D-INUNPACK-UNIQ

$$\frac{(\rho;\kappa;H;\iota.e) \to (\rho';\kappa';H';\iota.e';a)}{(\rho;\kappa;H;\iota.\texttt{inunpackuniq}(l;S;s)\ e) \to (\rho';\kappa';H';\iota.\texttt{inunpackuniq}(l;S;s)\ e';a)}$$

D-INUNPACK-IMM

$$\frac{(\rho;\kappa;H;\iota.e) \to (\rho';\kappa';H';\iota.e';a)}{(\rho;\kappa;H;\iota.\texttt{inunpackimm}(l;s)\ e) \to (\rho';\kappa';H';\iota.\texttt{inunpackimm}(l;s)\ e';a)}$$

D-INUNPACK-SHARE

$$\frac{(\rho;\kappa;H;\iota.e) \to (\rho';\kappa';H';\iota.e';a)}{(\rho;\kappa;H;\iota.\texttt{inunpackshare}(l;S;s)\ e) \to (\rho';\kappa';H';\iota.\texttt{inunpackshare}(l;S;s)\ e';a)}$$

D-UNPACK-UNIQ-LEAVE

$$\frac{H(o) = C(\overline{o})@\mathsf{up}}{(\rho;\kappa;H;\iota.\texttt{inunpackuniq}(l;S;s)\ o) \to (\rho;\kappa;H[\rho(l) \mapsto C(\overline{o})@s];\iota.o;\bullet)}$$

D-UNPACK-IMM-LEAVE

$$(\rho;\kappa;H;\iota.\texttt{inunpackimm}(l;s)\ o) \to (\rho;\kappa;H;\iota.o;\bullet)$$

D-UNPACK-SHARE-LEAVE

$$\frac{H(o) = C(\overline{o})@\mathsf{up}}{(\rho;\kappa;H;\iota.\texttt{inunpackshare}(l;S;s)\ o) \to (\rho;\kappa;H[\rho(l) \mapsto C(\overline{o})@s];\iota.o;\bullet)}$$

D-LET-E

$$\frac{(\rho;\kappa;H;\iota.e_1) \to (\rho';\kappa';H';\iota.e_1';a)}{(\rho;\kappa;H;\iota.\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \to (\rho';\kappa';H';\iota.\texttt{let}\ x = e_1'\ \texttt{in}\ e_2;a)}$$

D-LET-V

$$\frac{l \text{ fresh}}{(\rho;\kappa;H;\texttt{let}\ x = o\ \texttt{in}\ e) \to (\rho[l \mapsto o];\kappa;H;\iota.[x/l]e;\bullet)}$$

D-CALL

$$\frac{\mathsf{mdecl}(C,m) = E\ m(\overline{E \gg p\ x})[p \gg p'] = e}{(\rho;\kappa;H;\iota.l.m(\bar{l})) \to (\rho;\kappa;H;\iota.[\overline{l/x}][l/\texttt{this}]e;\bullet)}$$

D-SPAWN

$$(\rho;\kappa;H;\iota.\texttt{spawn}\ (l.m(\bar{l}))) \to (\rho;\kappa;H;\iota.\texttt{new}\ \mathsf{Object}();l.m(\bar{l}))$$

D-NEW

$$\frac{o \text{ fresh} \qquad \mathsf{init}(C) = \langle \overline{f : C.p}, s\rangle}{(\rho;\kappa;H;\iota.\texttt{new}\ C(\bar{l})) \to (\rho;\kappa;H[o \mapsto C(\overline{\rho(l)})@s];\iota.o;\bullet)}$$

### 4.2.7 Proof Judgments

So far we have presented the judgments and rules defining the language itself. Now we will present a number of judgments that will be useful in the proof of soundness for that language.

**Miscellaneous Context Judgments** This first judgment defines a notion of substitution for packing contexts. We will need such a notion when, in the proof, we perform substitution on an expression and would like the packing context to be appropriate for the new expression. Substitution only has an effect on unpacked contexts, and even then only on the term that denotes the currently unpacked object.

$$\boxed{[t_2/t_1]u}$$

$$\begin{array}{c}
[t_2/t_1]\mathsf{p} = \mathsf{p} \\
[t_2/t_1]\mathsf{up}(t_1; k; S; \overline{f : p}) = \mathsf{up}(t_2; k; S; \overline{f : p}) \\
[t_2/t_1]\mathsf{up}(t; k; S; \overline{f : p}) = \mathsf{up}(t; k; S; \overline{f : p})
\end{array}$$

In the theorem of single-threaded preservation, we need to mandate that the output linear context after a step is "bigger" than the output linear context before the step. By bigger we mean that anything one context can be used to prove, the other can be used to prove, and possibly more. This is not merely a matter of having permissions for all the same labels, but also having equal or stronger permissions in the bigger context. The following judgment defines just such a notion.

$$\boxed{\Delta \leq \Delta'}$$

$$\frac{\Delta \leq \Delta' \qquad p_1 \vdash \ p_2}{\Delta, b{:}p_1 \leq \Delta', b{:}p_2} \qquad \bullet \leq \Delta \qquad \frac{\Delta' \leq \Delta}{\Delta > \Delta'}$$

Likewise, in our proof of preservation we will need to show that one packing context is stronger than another, meaning it can be used to prove anything that the weaker context can. The following judgment defines such a notion.

$$\boxed{u \leq u'}$$

$$\frac{\overline{f : p} \vdash \ \overline{f' : p'}}{\mathsf{up}(t; k; S; \overline{f : p}) \leq \mathsf{up}(t; k; S; \overline{f' : p'})} \qquad \mathsf{p} \leq \mathsf{p} \qquad \frac{u' \leq u}{u > u'}$$

Note that this rule is really about the strength of the field permissions in the unpacked packing flag. The packed packing flag and the unpacked packing flag are incomparable. The same thing goes for two packing flags that differ by term $t$ or unpacking permission $k$.

In many of the following judgments (e.g., the important permsConsistent judgment which ensures that all permissions in a linear context are consistent with one another) we need to gather together all of the permissions in a given context or set of contexts. The following functions each generate a set of permissions associated with a given object $o$ from various contexts (where $+\!\!+$ stands for list concatenation):

$$
\begin{array}{lcl}
\mathsf{ctxPerms}(\Delta, o) & = & [p \mid o{:}p \in \Delta] \\
\mathsf{envPerms}(\Delta, \rho, o) & = & +\!\!+_{l \in \mathrm{dom}(\rho)[p \mid \rho(l)=o \text{ and } l:p \in \Delta]} \\
\mathsf{unpackedFields}(H, U, \rho, o) & = & \\
\multicolumn{3}{r}{+\!\!+_{o' \in \mathrm{dom}(H)}[p_i \mid u \in \mathrm{range}(U), u = \mathsf{up}(l; k; S; \overline{f:p}), \rho(l) = o', H(o') = C(\overline{o})\ \mathsf{up}, o_j = o]} \\
\mathsf{packedFields}(H, o) & = & +\!\!+_{o' \in \mathrm{dom}(H)}[p_i \mid H(o') = C(\overline{o})@s, \mathsf{inv}_C(s) = \overline{f:p}, o_i = o] \\
\mathsf{readoPerms}(H, o) & = & \\
\multicolumn{3}{r}{+\!\!+_{o' \in \mathrm{dom}(H)}[p_i \mid H(o') = C(\overline{o})@\mathsf{ro}(s), \mathsf{purify}(\mathsf{inv}_C(s)) = \overline{f:p}, o_i = o]} \\
\mathsf{perms}(H, \Delta, \rho, U, o) & = & \mathsf{unpackedFields}(H, U, \rho, o) +\!\!+ \mathsf{packedFields}(H, o) \\
\multicolumn{3}{r}{+\!\!+ \mathsf{readoPerms}(H, o) +\!\!+ \mathsf{envPerms}(\Delta, \rho, o) +\!\!+ \mathsf{ctxPerms}(\Delta, o)}
\end{array}
$$

Now we will proceed to describe a number of judgments that will ensure that the heap itself is consistent with the static contexts that type an individual thread or that type an entire thread pool. An important consequence of these rules is that the things that individual threads expect to be true based on their static typing contexts will be true of the run-time state.

**Single Thread Heap Invariant** The first such judgment is the single thread heap invariant. This judgment defines all of the facts that a single thread will need to guarantee are true after a step is taken, given that they are true before the step. (This local heap invariant will complement a similar heap invariant for the entire thread pool.)

The local heap invariant asserts several things. It asserts that types in $\Gamma$ have the types we expect. It asserts that any lock a thread believes it holds it does in fact hold. It asserts that any object a thread believes statically to be in a certain state is in that state, and any object it believes to be unpacked is unpacked. Finally, it asserts that a thread can only have definite knowledge about the state of an object of share permission if it also holds the lock to that object, and that it only unpacks share permissions when it holds the lock. The top-level judgment, $\Gamma; \Delta; \Psi; u \vdash \ \iota; \rho; \kappa; H$, depends on a number of smaller pieces.

The first such judgment, $\Gamma; \rho \vdash \ H$ ok, says that types given to variables in the static context $\Gamma$ are consistent with the types given in the heap. In addition, all of the types of the fields of that object in the heap must have the static type given by localFields. This process is recursively checked for fields, so that their fields are correct as well, and implicitly bottoms-out when the same object is checked more than once.

$$
\frac{\Gamma; \rho \vdash \ H \text{ ok} \qquad H(o) = C(\overline{o})@\$ \qquad \mathsf{localFields}(C, f_i) = C_i \qquad o_i{:}C_i \vdash \ H \text{ ok}}{\Gamma, b{:}C; \rho \vdash \ H \text{ ok}}
\quad o = o' \text{ when } b = o' \text{ or } o = \rho(l) \text{ when } b = l
$$

$$
\bullet; \rho \vdash \ H \text{ ok}
$$

The next judgment, $\rho; \Psi; \kappa \vdash \ \iota$ ok, enforces static lock consistency. It says that any lock known to hold statically because it is present in $\Psi$, must actually be held by the current thread, as listed in the dynamic locking context $\kappa$.

$$\frac{l \in \Psi \Rightarrow \rho(l) \overset{i}{\mapsto} \iota \in \kappa}{\rho; \Psi; \kappa \vdash \quad \iota \ \mathsf{ok}}$$

Next, the judgment $\Delta; \rho \vdash \quad H$ ensures consistency between the heap and facts in the linear context. The rules defining the judgment say the following: For every unique permission in the linear context, the corresponding object in the heap must be in a state $s'$ that is consistent with the permission state. Additionally, since the object is packed, the state invariant for the current state must be correct, meaning that all the fields of the object are in the states defined by that state invariant. The rule for immutable permissions is identical, except that the object in memory may be in either the state $s$, or a state $\mathsf{ro}(s)$, where in both cases $s$ is consistent with the permission state $S$. (Recall that $\mathsf{ro}(s)$ is the special read-only state that immutable objects enter when they are unpacked, and in which they remain for the rest of the program's lifetime.) Additionally, since immutable objects can only be unpacked in a purified state, the fields in the heap need only be consistent with a purified notion of the current state invariant. Finally, the share case is quite simple. Because share permissions must be consistent even during concurrent modification, a subsequent judgment, sharePerms, will take care of the tricky details.

$$\frac{\begin{array}{c} o = o' \text{ when } b = o' \text{ or } o = \rho(l) \text{ when } b = l \\ H(o) = C(\overline{o})@s' \qquad S \leftrightarrow s' \qquad \mathsf{inv}_C(s') = \overline{f : p} \qquad \overline{o : p}; \rho \vdash \quad H \qquad \Delta; \rho \vdash \quad H \end{array}}{\Delta, b{:}\mathsf{unique}@S; \rho \vdash \quad H}$$

$$\frac{\begin{array}{c} o = o' \text{ when } b = o' \text{ or } o = \rho(l) \text{ when } b = l \qquad H(o) = C(\overline{o})@\$ \\ (\$ = \mathsf{ro}(s) \vee \$ = s) \wedge S \leftrightarrow s \qquad \mathsf{purify}(\mathsf{inv}_C(s)) = \overline{f : p} \qquad \overline{o : p}; \rho \vdash \quad H \qquad \Delta; \rho \vdash \quad H \end{array}}{\Delta, b{:}\mathsf{immutable}@S; \rho \vdash \quad H}$$

$$\frac{H(\overline{o}) = C(\overline{o})@\$ \qquad \Delta; \rho \vdash \quad H}{\Delta, b{:}\mathsf{share}@S; \rho \vdash \quad H}$$

Next, a judgment $u; \rho \vdash \quad H$ defines what it means for a static unpacking context to be consistent with the heap. The first case is trivial. Packed unpacking contexts are automatically consistent with the heap. However, if the context is unpacked, then whether or not it is consistent depends on the permission kind with which it was unpacked. For unpacked share and unique permissions, the unpacked object in the heap must currently be in the special up state. Additionally, the permissions available in the context to the unpacked fields, $\overline{p}$, must be consistent with the actual field objects $\overline{o}$, a check which is accomplished using the previously-defined consistency judgment $o_i : p_i; \rho \vdash \quad H$. For unpacked immutable permissions, the same requirements exist, and additionally the permissions available in the context to the unpacked fields must always be equivalent to the purified state invariant.

$$\mathsf{p};\rho \vdash \quad H \qquad \frac{k = \mathsf{unique} \vee k = \mathsf{share} \qquad H(\rho(l)) = C(\overline{o})@\mathsf{up} \qquad o_i : p_i; \rho \vdash \quad H}{\mathsf{up}(l; k; S; \overline{f : p}); \rho \vdash \quad H}$$

$$\frac{H(\rho(l)) = C(\overline{o})@\mathsf{ro}(s) \qquad o_i : p_i; \rho \vdash \quad H \qquad S \leftrightarrow s}{\mathsf{up}(l; \mathsf{immutable}; S; \mathsf{purify}(\mathsf{inv}_C(s))); \rho \vdash \quad H}$$

The next three judgments define special requirements for share permissions, which exist because share permissions are subject to concurrent modification. The first judgment, $\mathsf{sharePerms}(\Delta, \Psi, u, \rho, H)$, exists to enforce the basic consistency requirements on share permissions. This is shown in the first rule defining the judgment where the standard consistency judgment $b{:}\mathsf{unique}@S; \rho \vdash \quad H$ is deferred to. It "pretends" that the permission is actually unique, forcing the deferred judgment to be its most restrictive. Note, however, that this is the most restrictive case. It need not be applied if the permission is immutable or unique. That is handled by the third rule. It also need not apply if there is currently a share permission unpacked. It is a strange fact that when a share object is unpacked, the other share permissions need not represent the actual state of the heap. This was found to be true when doing the proof the first time. Because of the possibility of other aliases to the same object existing in the same thread, it is difficult to formulate an otherwise satisfying consistency invariant. Note, however, that when a share permission is unpacked, using rule P-UNPACK-SHARE, all remaining share permissions in the linear context are downgraded, meaning that they will be consistent with the heap.

$$\frac{b{:}\mathsf{unique}@S; \rho \vdash \quad H \qquad \mathsf{sharePerms}(\Delta, \Psi, u, \rho, H)}{\mathsf{sharePerms}(\Delta, b{:}\mathsf{share}@S, \Psi, b, u, \rho, H)}$$

$$\frac{u = \mathsf{up}(l'; \mathsf{share}; S; \overline{f : p}) \vee b \notin \Psi}{\mathsf{sharePerms}(\Delta, b{:}\mathsf{share}@S, \Psi, u, \rho, H)}$$

$$\frac{k = \mathsf{unique} \vee k = \mathsf{immutable} \qquad \mathsf{sharePerms}(\Delta, \Psi, \rho, H)}{\mathsf{sharePerms}(\Delta, b{:}k@S, \Psi, u, \rho, H)}$$

The next judgment, shareLocks, ensures that all share permissions in the linear context that have a definite state $s$, correspond to objects for which the lock is held in the locking context. (Which we know, according to the judgment $\rho; \Psi; \kappa \vdash \quad \iota$ ok corresponds to the dynamic locking context.)

$$\frac{\{o \mid o : \mathsf{share}@s \in \Delta \vee l : \mathsf{share}@s \in \Delta \wedge o = \rho(l)\} \subseteq \{o \mid o = \rho(l) \wedge l \in \Psi\}}{\mathsf{shareLocks}(\Delta, \Psi, \rho)}$$

The judgment $\mathsf{shareUnpack}(u; \Psi; \rho)$ ensures that every unpacked share permission is for an object for which the lock is known to be held by the current thread.

$$\mathsf{shareUnpack}(\mathsf{p}, \Psi, \rho) \qquad \frac{k = \mathsf{unique} \vee k = \mathsf{immutable}}{\mathsf{shareUnpack}(\mathsf{up}(l; k; S; \overline{f : p}), \Psi, \rho)}$$

$$\frac{\exists l' \in \Psi \text{s.t.} \rho(l) = \rho(l')}{\mathsf{shareUnpack}(\mathsf{up}(l; \mathsf{share}; S; \overline{f : p}), \Psi, \rho)}$$

The judgment permsConsistent will allow us to easily say all permissions to the same objects consistent with one another. It is the last judgment that must be defined before we can present the single thread heap invariant. This judgment delegates to a judgment $\Delta; U; \rho; H \vdash \quad o$ ok, which says that all permissions $k_i$ to the same object $o$ are consistent, for which it depends upon the previously-defined consistency judgment, $k_i \leftrightarrow k_j$. Moreover, it does so for all permissions in all static contexts, which is done through use of the perms function. While this judgment is defined over linear context and unpacking *maps* ($\delta$ and U) it will still be useful for single contexts.

$$\boxed{\mathsf{permsConsistent}(\delta, U, \rho, H)}$$

$$\frac{\mathsf{perms}(H, \Delta, \rho, U, o) = \overline{k@S} \qquad \forall k_i \in \overline{k} \forall k_j \in \overline{k} \text{ where } i \neq j, \langle k_i \leftrightarrow k_j \rangle}{\Delta; U; \rho; H \vdash \quad o \text{ ok}}$$

$$\frac{\Delta = +\!\!+_{\Delta' \in \mathsf{range}(\delta)} \Delta' \qquad \Delta; U; \rho; H \vdash \quad \mathrm{dom}(H) \text{ ok}}{\mathsf{permsConsistent}(\delta, U, \rho, H)}$$

Lastly, we present the heap invariant for single threads. It uses each of the judgments just presented. To recap, given a set of static contexts and dynamic contexts, the single thread heap invariant holds if all of the types in $\Gamma$ are consistent with the heap, all locks in $\Psi$ are consistent with $\kappa$, permissions accurately reflect the states of objects in the heap, the unpacking flag accurately reflects the state of any unpacked object in the heap, share permissions are consistent with the heap when necessary, and locks are held to them if they mention definite states or if they are unpacked, and finally all permissions in all of the static contexts are consistent.

$$\boxed{\Gamma; \Delta; \Psi; u \vdash \quad \iota; \rho; \kappa; H}$$

LOCAL HEAP INV.

$$\frac{\begin{array}{c}\Gamma; \rho \vdash \quad H \text{ ok} \qquad \rho; \Psi; \kappa \vdash \quad \iota \text{ ok} \\ \Delta; \rho \vdash \quad H \qquad u; \rho \vdash \quad H \qquad \mathsf{sharePerms}(\Delta, \Psi, u, \rho, H) \qquad \mathsf{shareLocks}(\Delta, \Psi, \rho) \\ \mathsf{shareUnpack}(u, \Psi, \rho) \qquad \mathsf{permsConsistent}(\{\iota \mapsto \Delta\}, \{\iota \mapsto u\}, \rho, H)\end{array}}{\Gamma; \Delta; \Psi; u \vdash \quad \iota; \rho; \kappa; H}$$

**Top-Level Heap Invariant** The next judgment, that of a well-typed heap with respect to all the static contexts for every thread, both ensures consistency amongst all the static contexts and delegates to the single-threaded heap invariant. As before, we need to first define several helper judgments before we can present the invariant itself.

First there is the statesConsistent judgment which, much like the permsConsistent judgment in the previous section, ensures that all states mentioned by all permissions in all static contexts have states that are consistent. In other words, every permission in the linear context, unpacking context and from the state invariants of packed objects must have the same state or at least be ambivalent about the state ($?$).

$$\boxed{\mathsf{statesConsistent}(\delta, U, \rho, H)}$$

$$\frac{\mathsf{perms}(H, \Delta, \rho, U, o) = \overline{k@S} \qquad \forall S_i \in \overline{S} \forall S_j \in \overline{S}, \langle S_i \leftrightarrow S_j \rangle}{\Delta; U; \rho; H \vdash \quad o\ \mathsf{ok}}$$

$$\frac{\Delta = +\!\!+_{\Delta' \in \mathrm{range}(\delta)} \Delta' \qquad \Delta; U; \rho; H \vdash \quad \mathrm{dom}(H)\ \mathsf{ok}}{\mathsf{statesConsistent}(\delta, U, \rho, H)}$$

Next, we want to be sure that, statically, there is no overlap between the locks held by each thread. The judgment disjointLocks ensures this. The formalism is complicated, but it says a very simple thing: for every pair of locking contexts in a thread pool, there must be no overlap in the locks that they claim to hold.

$$\boxed{\mathsf{disjointLocks}(\psi, \rho)}$$

$$\mathsf{objects}(\Psi, \rho) = \{o \mid o = \rho(l), l \in \Psi\}$$

$\mathsf{disjointLocks}(\psi, \rho)$ true if
$\emptyset = \bigcup_{\Psi_i \in \mathrm{range}(\psi)} \mathsf{objects}(\Psi_i, \rho) \cap (\bigcup_{\Psi_j \in \mathrm{range}(\psi)/\Psi_i} \mathsf{objects}(\Psi_j, \rho)$

Similarly, the set of mutable objects unpacked by each thread must be disjoint.

$$\boxed{\mathsf{disjointUnpack}(U, \rho)}$$

$$\begin{array}{l}\mathsf{mutable}(\mathsf{p}, \rho) = \emptyset \\ \mathsf{mutable}(\mathsf{up}(l; \mathsf{unique}; S; \overline{f : p}), \rho) = \{\rho(l)\} \\ \mathsf{mutable}(\mathsf{up}(l; \mathsf{share}; S; \overline{f : p}), \rho) = \{\rho(l)\} \\ \mathsf{mutable}(\mathsf{up}(l; \mathsf{immutable}; S; \overline{f : p}), \rho) = \emptyset\end{array}$$

$\mathsf{disjointUnpack}(U, \rho)$ true if
$\emptyset = \bigcup_{u_i \in \mathrm{range}(U)} \mathsf{mutable}(u_i, \rho) \cap (\bigcup_{u_j \in \mathrm{range}(U)/u_i} \mathsf{mutable}(u_k, \rho)$

Now we can present the global heap invariant:

$$\boxed{\gamma; \delta; \psi; U \vdash \quad \rho; \kappa; H}$$

$$\frac{\mathsf{statesConsistent}(\delta, U, \rho, H) \qquad \mathsf{permsConsistent}(\delta, U, \rho, H) \qquad \mathsf{disjointLocks}(\psi, \rho) \qquad \mathsf{disjointUnpack}(U, \rho) \qquad \text{forall } \iota \in \mathrm{dom}(\gamma),\ \gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \quad \iota; \rho; \kappa; H}{\gamma; \delta; \psi; U \vdash \quad \rho; \kappa; H}$$

In other words, the permissions and permission states that each thread depends on must be consistent with one-another, the locks that they depend on must be disjoint, and the objects that they depend on being unpacked must be disjoint. Additionally, the single-thread heap invariant (previously discussed) must hold for every thread.

### Stack Invariants

Finally, there is a predicate relating the expression "stack" to the run-time state that must hold before and after each step of a thread.

Most of the properties enforced by the stackWF predicate are used in one or two places in the proof, but they all share the property that they rely on the expression in some way, typically the number of `insync` or `inunpackuniq` expressions that are subexpressions of the current thread. As before, the stackWF judgment depends on a number of smaller judgments.

First there is the nestingShare judgment. It examines an expression for a particular run-time configuration: the one in which an `inunpackshare` expression is wrapped in an `insync` expression that holds the lock for the object being unpacked. When this expression does not hold, we can deduce that the number of occurrences of a lock in $\Psi$ is one greater than the number appearing in an expression. Critically, this is needed in the D-Sync-Release-I case of the preservation proof.

| | | |
|---|---|---|
| $\mathsf{nestingShare}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$ | true if | $\mathsf{nestingShare}(e_1)$ |
| $\mathsf{nestingShare}(\texttt{insync}(l)\ e)$ | true if | $(\mathsf{activeUnpack}(e, \mathsf{share}) = \{l\}) \vee \mathsf{nestingShare}(e)$ |
| $\mathsf{nestingShare}(e)$ | true if | All other cases |

The judgment $\mathsf{upProtect}(\Psi, u, e)$ ensures that if the static packing flag says an object of share permission is unpacked, then a lock for that object is in the static lock context.

$$\begin{array}{lcl} \mathsf{occurrences}(\Psi, l) & = & |[l' \mid l' = l, l' \in \Psi]| \\ \mathsf{occurrences}(\Psi, \rho, o) & = & |[l' \mid \rho(l') = o, l' \in \Psi]| \end{array}$$

$$\frac{\mathsf{nestingShare}(e) \vee (u = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p}) \Rightarrow \mathsf{occurrences}(\Psi, l) > \mathsf{numLocks}(e, l))}{\mathsf{upProtect}(\Psi, u, e)}$$

The next judgment, $\Psi; e; \rho; \kappa; \iota \vdash \quad o\ \mathsf{ok}$, tells us that the locking context will always contain more locks for an object than the number of `insync` expressions, and that it in turn is less than or equal to the number of locks actually held by the thread in the run-time state.

$$\frac{\begin{array}{c}\mathsf{numLocks}(e,\rho,o) \leq \mathsf{occurrences}(\Psi,\rho,o) \leq i \\ (\forall l\ s.t.\ \rho(l) = o, \mathsf{numLocks}(e,l) \leq \mathsf{occurrences}(\Psi,l) \leq \mathsf{occurrences}(\Psi,\rho,o) \leq i) \\ \text{where } o \overset{i}{\mapsto} \iota \in \kappa\end{array}}{\Psi; e; \rho; \kappa; \iota \vdash \ o\ \mathsf{ok}}$$

The stack well-formedness judgment is defined as follows:

$$\boxed{\mathsf{stackWF}(\Psi, u, e, \rho, \kappa, \iota)}$$

$$\frac{\mathsf{activeLocks}(e) = s_2 \qquad \begin{array}{c}\mathsf{activeUnpack}(e) = s_1 \\ \Psi; e; \rho; \kappa; \iota \vdash \ \mathrm{dom}(\kappa)\ \mathsf{ok}\end{array} \qquad \mathsf{upProtect}(\Psi, u, e)}{\mathsf{stackWF}(\Psi, u, e, \rho, \kappa, \iota)}$$

In addition to ensuring the properties defined by the helper judgments, it also ensures that activeLocks and activeUnpack are well-defined, although it puts no restrictions on the sets, $s_1$ and $s_2$, defined by those function.

Throughout this section we presented a number of judgments describing thread and thread-pool well-formedness. Each one will need to be proven true in our proof of progress at every step of the computation, described in the following section.

## 4.3 Theorems and Proofs

Armed with both our top-level heap invariant and our thread heap invariant, we can now prove safety for our language. This will be done in two phases, first for the top level and then for the individual threads.

### 4.3.1 Top-Level Proof of Safety

**Theorem 1** (Top-Level Progress)**.** *Given a well-typed thread pool and an initial program state consistent with the typing of the pool (i.e., $\gamma; \delta; \psi; U \vdash \ L : \epsilon \ \dashv \delta_o; U_o$ and $\gamma; \delta; \psi; U \vdash \ \rho; \kappa; H$) Then it is the case that either all threads in $L$ are values or the entire thread pool can take a step.*

*Proof.* By induction on the single typing derivation for thread-pools.

**Case** (Well-Typed Pool)**.**

For a well-typed thread-pool, one of two states can hold for the expressions in the thread-pool. It it either the case that all threads are values $\iota.v$ or there is at least one thread in the thread-pool such that it is not a value. If all threads are values, then we are done. If, however, there is at least one that is not a value, the fact that the thread-pool is well-typed tells us that this particular thread, $\iota.e$ is also well-typed (i.e., $\gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \ e : E \ \dashv \delta_o(\iota); U_o(\iota)$). Moreover, the fact that

the initial program state is consistent with the static typing contexts tells us that it is also consistent with the specific typing contexts used to type $\iota.e$ (i.e., $\gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \quad \rho; \kappa; H$). Both of these facts come from the definitions of the predicates themselves. Given this, then by the theorem of thread-level Progress we know that $e$ can take a step to some expression $e$, according to the following judgment: $(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H'; \iota.e'; a)$.

Now, $a$ can have one of two values, $\bullet$ or $e_2$. If the former, then rule D-TOP-NORMAL applies, allowing the entire thread pool to take a step, $(\rho; \kappa; H; L, \iota.e) \rightarrow (\rho'; \kappa'; H'; L, \iota.e')$. Otherwise, the rule D-TOP-SPAWN applies, allowing the entire thread tool to take a step, $(\rho; \kappa; H; L, \iota.e) \rightarrow (\rho'; \kappa'; H'; L, \iota.e', \hat{\iota}.e_2)$.

□

**Theorem 2** (Top-Level Preservation)**.** *Given a well-typed thread pool in a consistent program state that takes a step (i.e., $\gamma; \delta; \psi; U \vdash \quad L : \epsilon \quad \dashv \delta_o; U_o$ and $\gamma; \delta; \psi; U \vdash \quad \rho; \kappa; H$ and $(\rho; \kappa; H; L) \rightarrow (\rho'; \kappa'; H'; L')$), there exists static typing contexts $\gamma', \delta', \psi', U'$ and new types $\epsilon'$ such that the resulting thread pool is well-typed and consistent with the new program state (i.e., $\gamma'; \delta'; \psi'; U' \vdash \quad L' : \epsilon' \quad \dashv \delta'_o; U'_o$ and $\gamma'; \delta'; \psi'; U' \vdash \quad \rho'; \kappa'; H'$).*

*Proof.* By induction on the top-level dynamic rules.

**Case** (D-Top-Normal)**.**

The proof for this case and the next case are quite similar. We are given that the thread pool is well-typed and that its typing contexts are consistent with the run-time state.

By the definition of a well-typed thread pool, $\gamma; \delta; \psi; U \vdash \quad L, \iota.e : \epsilon \quad \dashv \delta_o; U_o$, we know that the single thread $e$ is also well-typed, $\gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \quad e : \epsilon(\iota) \quad \dashv \delta_o(\iota); U(\iota)$. By the top-level heap invariant, which we are given holds, we know that the single thread heap invariant holds for the contexts under which $e$ is well-typed, $\gamma(\iota); \delta(\iota); \psi(\iota); U(\iota) \vdash \quad \iota; \rho; \kappa; H$. Finally, the premise of D-TOP-NORMAL tells us that the thread can take a step $(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H'; \iota.e'; \bullet)$.

The theorem of single-threaded Preservation tells us that, given these facts, $e'$ can be well-typed under new static contexts, and those contexts are consistent with the resulting run-time state (i.e., $\Gamma'; \Delta'; \Psi'; u' \vdash \quad e' : E' \quad \dashv \Delta'_o; u'_o$ and $\Gamma'; \Delta'; \Psi'; u' \vdash \quad \iota; \rho'; \kappa'; H'$).

Therefore, let us choose the resulting static contexts for the entire thread-pool to be the original $\gamma, \delta, \psi, U$ and the types to be $\epsilon$ except modified so that $\gamma(\iota) = \Gamma', \delta(\iota) = \Delta', \psi(\iota) = \Psi', U(\iota) = u', \epsilon(\iota) = E'$. We will call these new context maps, $\gamma', \delta', \psi', U'$ and the new types $\epsilon'$.

Now, we must show that $\gamma'; \delta'; \psi'; U' \vdash \quad L, \iota.e' : \epsilon' \quad \dashv \delta'_o; U'_o$ and $\gamma'; \delta'; \psi'; U' \vdash \quad \rho'; \kappa'; H'$.

The first part, that the thread-pool is still well-typed, is easy. The definition of a well-typed thread pool is simply that all of the threads are well-typed. All threads were well-typed before the step, and the only static contexts that we changed were for the changed thread $\iota.e'$. However, the result of our use of the single-threaded Preservation theorem is that $e'$ is well-typed under its new static contexts, therefore the entire thread-pool must be well-typed under these new contexts.

Showing that the global heap invariant still holds is more involved. There are four important predicates that we must prove still hold for the new static contexts $\gamma', \delta', \psi', U'$ with respect to the new global state $\rho', \kappa', H'$. First, we must show that the statesConsistent$(\delta', U', \rho', H')$ predicate holds. This predicate is dependent on the states of all the permissions in $\delta', u'$ and for

every packed object, on the declared state invariant. During one step of a single thread, the single-threaded progress theorem makes guarantees about what a thread will do to the permissions in its contexts and in the heap. As far as this predicate is concerned, a thread step can do one of four things.

1. It can move a permission, either from the heap $H$ to the unpacking flag $u'$, or from the unpacking flag $u$ to the linear context $\Delta$, including the possibility of a packed/unpacked object.
2. It can create a new permission for an entirely new heap object $o$ in $\Delta'$.
3. It can change the state of a permission in $\Delta$ if the permission is unique or share.
4. It can drop a permission altogether.

None of these actions will invalidate the predicate statesConsistent, which was true before the step. In the first action, if a permission is moved, it does not change the state, and therefore a state that was consistent with all the other states will remain so. Second, a completely new object cannot be mentioned in the statesConsistent predicate before the step, and therefore cannot conflict with other permissions after the step, since there are none. If the state of an object of unique permission is changed, it could not have existed in any other static context $\Delta'$ before the step, and therefore there are no other permission states to invalidate. If the state of a share permission changes, the single-threaded Preservation theorem guarantees that that particular thread must have had $l \in \Psi$ such that $\rho(l) = o$, for the object under discussion. disjointLocks, which was true before the step, guarantees that no other thread can have a lock to the same object in its static contexts. Combine this with the fact that we are guaranteed by the single-threaded heap invariant, which was true for all threads before the step, that a precise share permission cannot exist in $\Delta$ unless a lock for that object is in the lock context, and we know that any other share permissions to the same object must be imprecise ($?$), which is consistent with all states. (Additionally, we know that state invariants and unpacking flags cannot hold precise share permissions.) Finally, if a permission is dropped all-together, it has no state to conflict with the other permissions.

A very similar series of reasoning steps applies to prove that permsConsistent$(\delta', U', \rho', H')$ after the single thread step. A thread step can only do one of four things to the permission kind:

1. It can move a permission, either from the heap $H$ to the unpacking flag $u'$ or from the unpacking flag $u$ to the linear context $\Delta$, including the possibility of a packed/unpacked object.
2. It can create a new permission in for an entirely new heap object $o$ in $\Delta'$.
3. It can split a permission using the splitting rules.
4. It can drop a permission altogether.

Again, if permissions are merely moved, permsConsistent will continue to hold. For new permissions to new objects, it holds because no other static context or packing flag could have possibly held any permission to the object, so there is no room for inconsistency. Permission consistency continues to hold even under splitting, which can be seen by examining the definition of splitting and the definition of consistency. And finally, dropped permissions cannot conflict with any other permission.

If $\mathsf{disjointLocks}(\psi, \rho)$ holds before the single thread step, it will hold after. This is because, according to the single-threaded heap invariants, which hold for all threads before the step, if $l \in \Psi$ then $\rho(l) \overset{i}{\mapsto} \iota \in \kappa$. If $\Psi'$ for the stepping thread contains an $l$ that was not in $\Psi$, it will only do so based on the knowledge that no lock is held on $\rho(l)$ in $\kappa$. This can be seen by examining the lock acquisition rules for single threads. So, if $o \overset{i}{\mapsto} \iota$ is not in $\kappa$ before the step, for some $\iota$, then there could not have been any $l$ in all $\psi$ such that $\rho(l) = o$.

$\mathsf{disjointUnpack}(U', \rho')$ also holds for similar reasons. Again, the single-thread heap invariant, which must be true before the step according to the global heap invariant, says that if a packing flag $u$ for any thread is unpacked, then the heap must reflect this (e.g., $H(\rho(l)) = C(\overline{o})@\mathsf{up}$ for share and unique permissions or $H(\rho(l)) = C(\overline{o})@\mathsf{ro}(s)$ for immutable permissions). In the first case, $u' = \mathsf{p}$, in which case the single thread step could not have possibly introduced an inconsistency. However, if $u' = \mathsf{up}(l; k; S; \overline{f : p})$, then there are two cases depending on $k$. If $k = \mathsf{immutable}$, then the thread $e$ could have only unpacked the object if the previous heap state for $\rho(l)$ were $\mathsf{ro}(s)$ or $s$. This we can prove to ourselves by looking at each of the single-threaded dynamic rules that unpack an object. If other threads had unpacked the same object before the step, they must have done so with immutable permission, since this immutable permission cannot co-exist with others by permission consistency. Those other unpacked permissions must have $u_i = \mathsf{up}(l'; \mathsf{immutable}; s; \overline{f : p})$ which is compatible with $\mathsf{ro}(s)$, the heap state after the step. If instead, the unpacked object were unpacked with share permission, it must have had the lock for that object, by the single-threaded heap invariant which is true after the step, which in turn means that it could not be unpacked in any other thread. Therefore it could not conflict. Finally, if the thread unpacked the object with unique permission then it could not have been unpacked before the step by any other thread, since unique permission cannot coexist with any other permissions.

This satisfies four of the five premises of the global heap invariant, and, we know that the single-threaded heap invariant must be true for the thread that took a step. However, we also must show that the single-threaded heap invariants remain true for all of the other threads that did not take a step. This largely involves reasoning analogous to that which we have already applied. For all of the threads that did not take a step, here is how we know the single-threaded heap invariant still holds (these will be presented in the order of the premises of rule LOCAL HEAP INV.):

First, the types in $\gamma(\hat{\iota})$, where $\hat{\iota}$ is the thread identifier of any thread that did not take a step, are still correct. By Lemma 1, taking a step will not change the class $C$ of any location that was in the heap before the step.

Second, any lock $l \in \psi(\hat{\iota})$ before the step, which guaranteed that $\rho(l) \overset{i}{\mapsto} \hat{\iota} \in \kappa$ remains true, since as we previously mentioned, none of the dynamic rules allow a thread to take a lock mapped to a different thread.

Next, for all threads that did not take a step, the permissions contained in $\delta(\hat{\iota})$ must still be consistent with the heap. We can prove this by appealing to the same reasoning used for statesConsistent and permsConsistent. Single threads cannot increase the amount of permission that they have to an object and they cannot change the state of an object except for objects to which they have unique permission or to share objects, for which they must hold locks. By holding a lock to share objects, it is implied that all other threads must have had share@? permissions, if they hold any permission at all, meaning their permissions are still consistent with the state. unique permissions cannot be shared between threads.

The consistency of a thread's packing flag $U(\hat{\imath})$ with the heap also cannot be destroyed by a stepping of another thread. If $U(\hat{\imath}) = \mathsf{p}$ before the step, then it will still be consistent. For immutable unpacks, $U(\hat{\imath}) = \mathsf{up}(l; \mathsf{immutable}; s; \overline{f : p})$, the stepping thread could have at most immutable permission, which it cannot use to change the object in the heap. For unique unpacks, $U(\hat{\imath}) = \mathsf{up}(l; \mathsf{unique}; S; \overline{f : p})$, the stepping thread could not change the state of the object in memory at all, since it could not have permission. And for share unpacks, $U(\hat{\imath}) = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p})$, the thread $\hat{\imath}$ must have the lock, which prevents the stepping thread from unpacking and modifying the object.

For the threads that do not take a step, the shareLocks and shareUnpack predicates continue to hold, since they depend only on static information, which has not changed, and $\rho'$ for which existing mappings cannot change.

This means that the local heap invariant holds for all threads after the step, which means that the global heap invariant holds.

**Case** (D-Top-Spawn)**.**

This case is largely identical to the previous case except we must show that the newly spawned thread can be well-typed in some static contexts and that those contexts are consistent with the existing thread pool.

By the single-threaded Preservation theorem, if a single thread takes a step, and spawns a thread as a side-effect, $(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H; \iota.e'; e_2)$, then it cannot change the run-time state, so $\rho' = \rho, \kappa' = \kappa, H' = H$. Moreover, if the thread was well-typed before the step, then after the step the following typing judgments must hold: $\Gamma; \Delta_1; \Psi; u \vdash \quad e' : E \quad \dashv \Delta_o; u_o$ and $\Gamma; \Delta_2; \bullet; \mathsf{p} \vdash \quad e_2 : E_2 \quad \dashv \Delta'_o; u'_o$ where $D \vdash \quad \Delta_1$, $D \vdash \quad \Delta_2$ and $\mathsf{permsConsistent}(\Delta_1, \Delta_2)$. So in other words the original expression that has taken a step must remain typeable in its old static contexts and the new thread must be typeable in a context with no locks and no unpacked objects and with the same valid context. The only difference is, both threads must "share" the linear context that typed $e$ before the step.

Therefore, because all the static contexts were consistent with the heap and with the other threads before the step, and the resulting static contexts are essentially unchanged with the exception of $\Delta$ being split between two threads, all of the predicates that were true before the step are still true. □

### 4.3.2 Single-Threaded Guarantees

The lemma and predicates in this section are used by the top-level proof of preservation and established during the execution of a single-thread. They ensure that the execution of one thread will not violate the expectations of any other thread.

**Lemma 1** (No Class Changes)**.** *If a thread can take a step from a run-time state, and $H(o) = C(\overline{o})@\$$ for some object in the heap before the step, then $H'(o) = C(\overline{o'})@\$'$. In other words, the thread will not change the class of an existing object.*

*Proof.* By induction over the cases of the transition relationship. No transitions change the class of an object in the heap, they will only ever create new objects. □

The guaranteed judgment which follows will be used in the single thread proof of preservation. Unlike the lemma above, this judgment must be shown to hold at each step of the preservation proof. This judgment defines the guarantees that a single thread makes to the rest of the thread pool. These guarantees are two-fold. First, a single thread guarantees that it will not make permissions up out of nowhere, it will only move them from one static context to the next. This is the guarantee defined by the judgment movement, and which is defined informally.

$$\boxed{\mathsf{movement}(\Delta; u; \rho; H; \Delta'; u'; \rho'; H')}$$

- Any permission in $\Delta'$ that was not present in $\Delta$ is either for a completely new heap object, was removed/split from $u$ or is for a just-packed object.
- Any field permission in $u'$ that was not present in $u$ is either from a state invariant for a object that is unpacked in $H'$ or was removed/split from $\Delta$.
- Any packed object in $H'$ that was unpacked in $H$ must have been unpacked in the unpacking flag $u$ or is a completely new object whose field permissions come from permissions previously present in $\Delta$.

The second part of the single thread guarantee relates the two heaps, the heap before a step and after a step. This judgment, permsNeeded, shows that the state of an object in the heap will only be changed by a single thread if it actually had permission to that object, either in its linear context or in the unpacking context.

$$\frac{\begin{array}{c} H(o) = C(\overline{o}, o_i)@\$ \\ H'(o) = C(\overline{o}, o_i')@\$' \qquad \$ \neq \$' \text{ iff } (l{:}k@S \in \Delta \vee u = \mathsf{up}(l; k; S; \overline{f : p} \wedge u' = \mathsf{p}) \wedge \rho(l) = o \\ o_i \neq o_i' \text{ iff } u = \mathsf{up}(l; k; \overline{f : p}) \wedge (k = \mathsf{share} | k = \mathsf{unique}) \wedge \rho(l) = o \end{array}}{\mathsf{permsNeeded}(\Delta; u; u'; \rho; H; H')}$$

$$\boxed{\Delta; u; \rho; H \vdash \quad \Delta'; u'; \rho'; H' \text{ guaranteed}}$$

$$\frac{\mathsf{movement}(\Delta; u; \rho; H; \Delta'; u'; \rho'; H') \qquad \mathsf{permsNeeded}(\Delta; u; u'; \rho; H; H')}{\Delta; u; \rho; H \vdash \quad \Delta'; u'; \rho'; H' \text{ guaranteed}}$$

### 4.3.3 Thread-Level Proof of Safety

**Theorem 3** (Single-Threaded Progress)**.** *For any closed expression $e$ that is well typed in a program state such that the static contexts satisfy the single-threaded heap invariant, i.e.*

$$\Gamma; \Delta; \Psi; u \vdash \quad e : E \quad \dashv \Delta_o; u_o \textit{ and } \Gamma; \Delta; \Psi; u \vdash \quad \iota; \rho; \kappa; H,$$

*it is either the case that $e$ is a value, or $e$ can take a step to some other expression $e'$, i.e.*

$$(\rho; \kappa; H; \iota.e) \rightarrow (\rho'; \kappa'; H'; \iota.e'; a).$$

*Proof.* By induction over the cases of the typing judgment $\Gamma; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta_o; u_o$. See Section B.1 for the complete proof. □

**Theorem 4** (Single-Threaded Preservation)**.** *For any closed expression $e$ that is well-typed, and in a program state satisfying the single-threaded heap invariant, i.e.*

$$\Gamma; \Delta; \mathsf{activeLocks}(e), \Psi_2; u \vdash \ e : E \ \dashv \Delta_o; u_o \textit{ and } \Gamma; \Delta; \Psi; u \vdash \ \iota; \rho; \kappa; H,$$

*if that expression can take a step to another expression $e'$, via the judgment $(\rho; \kappa; H; \iota.e) \to (\rho'; \kappa'; H; \iota.e'; a)$, then there exist static typing contexts $\Gamma', \Delta', \Psi', u'$ and a type $E'$ such that this resulting expression is well typed and the static contexts satisfy the single-threaded heap invariant, i.e.*

$$\Gamma'; \Delta'; \Psi'; u' \vdash \ e' : E' \ \dashv \Delta'_o; u'_o \textit{ and } \Gamma'; \Delta'; \Psi'; u' \vdash \ \iota; \rho'; \kappa'; H'.$$

*Given that the "stack" is well-formed before the step it is well-formed afterward (i.e., given* $\mathsf{stackWF}(\Psi, u, e, \rho, \kappa, \iota)$ *then* $\mathsf{stackWF}(\Psi', u', e', \rho', \kappa', \iota)$*).*

*Next, during the step the thread does not violate any of its guarantees to other threads. Meaning, the judgment* $u; \Delta; H \vdash \ u'; \Delta'; H'$ $\mathsf{guaranteed}$ *holds.*

*The side-conditions on the static contexts hold:*

- $\downarrow(E) = \downarrow(E')$
- $\downarrow^{\Psi_2}(\Delta_o) \leq \downarrow^{\Psi_2}(\Delta'_o)$
- $u_o \leq u'_o$
- $\Gamma \leq \Gamma'$
- $\Psi' = \mathsf{activeLocks}(e'), \Psi_2$
- $\mathsf{activeUnpack}(e) = \emptyset \wedge u = \mathsf{up}(l; k; S; \overline{f : p}) \Rightarrow \mathsf{activeUnpack}(e') = \emptyset \wedge u' = \mathsf{up}(l; k; S; \overline{f : p})$

*And finally, if a thread is spawned ($a = e_2$) then it is possible to divide $\Delta$ into two parts such that $\Delta \vdash \ \Delta_1$ and $\Delta \vdash \ \Delta_2$ and $\Delta_1 = \Delta'$, and* $\mathsf{permsConsistent}(\Delta_1, \Delta_2)$*. $\Gamma' = \Gamma, u' = u, \Psi' = \Psi, H' = H, \rho' = \rho, \kappa' = \kappa$. And the new expression is well-typed $\Gamma; \Delta_2; \bullet; \mathsf{p} \vdash \ e_2 : E_2 \ \dashv \Delta'_o; u'_o$ such that $\Gamma; \Delta_2; \bullet; \mathsf{p} \vdash \ \hat{\iota}; \rho; \kappa; H$.*

*Proof.* By induction over the cases of the single-threaded step judgment,

$$(\rho; \kappa; H; \iota.e) \to (\rho'; \kappa'; H'; \iota.e'; a).$$

For the complete proof, see Section B.1. □

Finally, note that Section B.3 of the Appendix contains a number of common lemmas that are used in type safety proofs, for example Substitution, and Weakening.

# Chapter 5

# Polymorphic Access Permissions

*Faithless is he who quits when the road darkens.*

---

## 5.1 Introduction

In this chapter we will extend the type system presented in Chapter 3 with parametric polymorphism over access permissions. This feature is motivated by our experiences during many of the earliest attempts to use Sync-or-Swim. We noticed particular patterns and data structures that we were unable to specify, or at least to specify succinctly. Parametric polymorphism helps us address many of those challenges.

Polymoprhism over access permissions is important for many of the same reasons that parametric polymorphism is already useful in standard type systems; it increases the precision of the type system for types whose implementations are in some sense "ambivalent" about the objects they reference. In Java 1.5 we can use polymorphism (or "generics") to define a class `Stack<T>`, a stack that holds elements of any type. When that class is instantiated with some type, say `File`, we are ensured that the particular instance will only accept and return files.

In our case we are allowing polymorphism over access permissions. Access permissions statically describe the current state of an object reachable through a reference, and whether or not that reference may be aliased by other references. By enabling polymorphism over access permissions, programmers can write classes that are ambivalent about their elements' protocols and level of aliased-ness. Continuing our theme, the `Stack<T>` class can be given polymorphic permission specifications. Now, a stack can be instantiated in a variety of different ways, signifying that different instances hold objects of different permission. For example, one stack can hold unique pointers to open files, while another stack holds shared pointers to sockets guaranteed to be initialized.

To give a clearer picture of the difficulties that arise without polymorphism, we will attempt to specify just such a stack using our existing methodology. Figure 5.1 shows the implementation and specification of a mutable stack. This stack defines only two methods, `push` and `pop`. If `pop`

is called when the stack is empty, it simply returns `null`. The stack defines no protocol of any interest, but since it is generic in the types of the elements it holds, its elements very well might.

```
1  @Invariants(@State(name="alive", inv="unique(first) in alive"))
2  class Stack<T> {
3    @Invariants(@State(name="alive",
4      inv="unique(next) in alive * pure(item) in alive"))
5    class Node { T item; Node next; }
6    Node first;
7
8    @Spec(post="unique(this) in alive")
9    Stack() { first = null; }
10
11   @Spec(pre="unique(this) in alive * pure(item) in alive",
12         post="unique(this) in alive")
13   void push(T item)  { Node n = new Node();
14     n.item = item; n.next = first;
15     first = n;
16   }
17
18   @Spec(pre="unique(this) in alive",
19         post="unique(this) in alive * pure(result) in alive")
20   T pop() {
21     if( first == null ) return null;
22     else { T result = first.item;
23       first = first.next; return result;
24     }
25   }
26 }
```

Figure 5.1: A specification of a `Stack` class, without polymorphism. It is weak in the sense that no matter what state the elements of the stack are in, the caller of `pop` method only knows the element are in the "alive" state.

We have attempted to specify this stack in as general a way as possible. Because the implementation does not constrain the types of the elements it holds, it also does not constrain the protocols defined by those elements. If a programmer only pushes open files on the stack, he expects open files to be returned from the stack. Verification-wise, note that the implementation does not constrain the permission kind associated with those elements. In other words, any permission that the caller of the `push` method is willing to forfeit to the pushed item, can soundly be transferred to the eventual caller of the `pop` method. (The stack maintains no references to popped items). For these reasons, the access permissions to the stack elements (highlighted in bold) in Figure 5.1 are as general as possible: the element must be in the "alive" state (trivially satisfied by every object) and the element must have pure permission kind (satisfiable with any other permission kind).

Unfortunately, this specification is quite imprecise. While it is easy to satisfy the pre-condition of the `push` method (line 11), for any object, the post-condition of the `pop` method (line 19) is quite weak. For example, for a stack of type `Stack<File>`, even if we push an

open file, and the calling site has unique permission, this information is not reflect in the post-condition of the pop method. The caller of the pop method receives a simple pure permission to the file in the alive state. The caller would be unable to use methods that depend on the file being open, for example read. This imprecision is analogous to state of Java collections before generics; the return type of the pop method of Stack could only guarantee that the returned object was of type Object.

In our experience with Sync-or-Swim, and in earlier experiences with access permissions [16], we have seen that one is forced either to copy and re-specify an implementation of a collection several times, once for each context in which it is used, or settle for false-positives. Collections are ubiquitous in most applications, meaning that an inability to succinctly specify their behavior is a real limitation. In concurrent programs, queues and sets are frequently used as a means of ownership transfer between different threads.

Therefore we now present an extension to our existing type system that allows us to give Stack a single polymorphic specification. A stack of unique, open files can share an implementation with a stack of shared, open sockets without losing precision, such as in the specification of the pop method. This polymorphic type system was originally proposed as an extension to the single-threaded Plural methodology [15]. In this chapter we will largely ignore concurrency in order to focus on the interesting aspects of polymorphism.

This chapter will proceed in the following manner: Section 5.3 describes the type system in technical detail and contains the primary contribution of this work. As the section progresses, we will attempt to motivate our new features and present a number of useful examples. Since the specifications end up being rather verbose, Section 5.4 shows how we can introduce syntactic sugar that greatly decreases the size of specifications written by programmers. Section 5.5 describes additions made to Sync-or-Swim, our implementation, which allow us to gain the features of polymorphism. Finally, the chapter ends with a discussion of related work and a conclusion.

## 5.2 Overview

This section summarizes the effect that polymorphic access permissions has on our static analysis from the perspective of users of the Sync-or-Swim analysis. In the remainder of the chapter we will reintroduce polymorphic access permissions at a very low level of granularity in our formal system. The polymorphic permissions presented in this section are in fact syntactic sugar on top of the lower level system.

We modify the type system to allow polymorphic permission variables to be introduced at the scope of a class or method. Where permission variables are in scope, they can be used in pre- or post-condition specifications or in state invariants. At class instantiation sites and method call sites, these permission variables must be instantiated with a permission, and the permissions are substituted uniformly through the method and state specifications for that instance. The instantiating permissions must match the declared bound on the permission variables, of which there are three kinds, exact, similar and symmetric. A permission variable with the bound exact can be instantiated with any kind of permission, but each time is is then used it will have to be of the exact same fractions. Contrast this with similar permission variables, which once instantiated will match any permissions of the same kind regardless of fractional values. Finally, the symmetric

permission bound can only be instantiated with those kinds of permissions that can be split multiple times and still produce permissions of the same kind, namely share, immutable and pure. Such a variable bound allows for the implementation to freely split permission variables while remaining ambivalent as to their kind.

```
1  @Similar(type="T",var="p")
2  @Invariants(@State(name="alive", inv="unique(first) in alive"))
3  class Stack<T> {
4    @Invariants(@State(name="alive", inv="unique(next) in alive * p(item)"))
5    class Node { T item; Node next; }
6    Node first;
7
8    @Spec(post="unique(this) in alive")
9    Stack() { first = null; }
10
11   @Spec(pre="unique(this) in alive * p(item)",
12         post="unique(this) in alive")
13   void push(T item)  {
14     Node n = new Node();
15     n.item = item; n.next = first;
16     first = n;
17   }
18
19   @Spec(pre="unique(this) in alive",
20         post="unique(this) in alive * p(result)")
21   T pop() {
22     if( first == null ) return null;
23     else {
24       T result = first.item;
25       first = first.next;
26       return result;
27     }
28   }
29 }
```

Figure 5.2: A revised specification of the `Stack` class which takes advantage of polymorphic specifications

Figure 5.2 shows a revised specification of the `Stack` class that takes advantage of polymorphism. On line 1 a new polymorphic permission variable named `p` is introduced, and its bound is similar. The `push` method consumes a permission of kind `p` to the `item` parameter. The `pop` method returns one to its callers.

Such a permission becomes useful when it is instantiated at a client site, as in Figure 5.3. On line 1, a ground permission, share in the guaranteed Open state, is applied to the `Stack` type, much in the same way that the `File` type is applied to its parametric type. Now calls to the `push` method require share permission to a file in the Open state, and calls to `pop` return the same.

The `push` method can legally take a share permission of any fractional value, thanks to the similar bound on `p`. If the exact bound were used, every call to `push` would require a permission of the exact same fraction. While not useful in this case, such a bound may be helpful when

```
1  void foo(@Apply("share(Open)") Stack<File> files) {
2    File f = ...;
3    files.push(f); // only valid if share(f,Open) is available
4
5    ...
6
7    while( !files.isEmpty ) {
8      File tmp = files.pop();
9      ... // share(tmp,Open) is now available
10   }
11 }
```

Figure 5.3: A client-side use of a polymorphic stack

designing an application that needs to carefully recombine all permissions to an object in order to reacquire uniqueness. The symmetric bound would be useful if our stack allowed random access to its elements. Symmetric permissions, since they can be repeatedly divided, could be returned by each call to the method providing random access, given the additional restriction that the collection could not hold unique or full permissions.

Such a system of polymorphism greatly increases the expressiveness of Sync-or-Swim specifications at a low conceptual burden for programmers already familiar with Java Generics.

## 5.3 Polymorphic Access Permissions

This section describes our extension in technical detail. The basic idea is to take each element of the access permission and allow the programmer to abstract over it at the method and class levels. Bounds on these abstracted variables, enforced at instantiation-time, ensure that the well-formedness rules of the access permissions are respected. Additionally, and perhaps most interestingly, the system allows programmers to abstract over the *classifiers* of fractions and fraction functions, not just the fractions themselves. This is useful because it allows programmers to instantiate a collection with a permission kind (e.g., share) while remaining ambivalent about exact fraction values.

Figure 5.4 gives the basic syntax for our extended language. It is very similar to the syntax presented in Chapter 3, however it lacks concurrency constructs, and the ability to extend existing classes to create sub-types has been introduced. A few points are worth reiterating: Each class can declare a number of new states and dimensions, $R$. Dimensions, $d$, refine existing states by introducing a number of new, mutually exclusive sub-states. As previously mentioned, an object must always be in one state in each dimension that it defines. Any state or dimension can be associated with a predicate, called a state invariant, $N$, that must hold whenever the object is in that node. Field declarations $F$ declare that a field is "mapped" into a node, and that field can only be modified when the object is in that node. This helps ensure that state guarantees actually guarantee an object's concrete state. Every method has a specification, $MS$, which consists of a pre- and post-condition, showing which permissions it requires for the receiver and parameters, and which permissions it returns upon completion. State invariants and method

specifications are written using linearl logic predicates, $P$ (Figure 5.5). Access permissions $p$, mention fractions $k$ and fraction functions $g$. A fraction is a literal 0 or 1, or a fraction divided by two. A fraction function is the mapping of a node to a fraction, a fraction function divided by two or the concatenation of two fraction functions.

$$
\begin{array}{rrcl}
\textit{programs} & PR & ::= & \langle \overline{CL}, e \rangle \\
\textit{class decl.} & CL & ::= & \texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T}\rangle[\overline{a}]\ \{\ \overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\ \} \\
\textit{field decl.} & F & ::= & f : T\ \texttt{in}\ n \\
\textit{state decl.} & R & ::= & d = \overline{s}\ \texttt{refines}\ s_0 \\
\textit{initial state} & I & ::= & \texttt{initially}\ \langle P, s_1 \otimes \ldots \otimes s_n \rangle \\
\textit{state inv.} & N & ::= & n = P \\
\textit{meth. decl.} & M & ::= & T\ m[\overline{\alpha:\kappa}](\overline{T\ x}) : MS = e \\
\textit{meth. spec.} & MS & ::= & P \multimap E \\
\textit{terms} & t & ::= & x \mid \texttt{true} \mid \texttt{false} \\
 & & \mid & t_1\ \texttt{and}\ t_2 \mid t_1\ \texttt{or}\ t_2 \mid \texttt{not}\ t \\
\textit{expressions} & e & ::= & t \mid f \mid f := t \\
 & & \mid & \texttt{new}\ C\langle\overline{T}\rangle[\overline{a}](\overline{t}) \mid t_0.m[\overline{a}](\overline{t}) \mid \texttt{super}.m[\overline{a}](\overline{t}) \\
 & & \mid & \texttt{if}(t, e_1, e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
 & & \mid & \texttt{unpack}(n, k, A)\ \texttt{in}\ e \mid \texttt{pack}\ n\ \texttt{to}\ A\ \texttt{in}\ e \\
\textit{references} & r & ::= & x \mid f \\
\textit{types} & T & ::= & \texttt{bool} \mid \beta \mid C\langle\overline{T}\rangle[\overline{a}] \\
\textit{nodes} & n & ::= & \alpha \mid s \mid d
\end{array}
$$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *classes* | $C$ | *fields* | $f$ | *variables* | $x$ | | |
| *methods* | $m$ | *states* | $s$ | *dimensions* | $d$ | *type variables* | $\beta$ |

Figure 5.4: Syntax I: Programs, Classes, Terms and Expressions

### 5.3.1 The Syntax of Permissions and Abstraction

The most interesting new addition to the syntax is the ability to introduce type variables at the class level, and permission variables at the class and method level. As seen in Figure 5.4, a class can introduce any number of type variables, $\beta$. Type variables allow classes to be generic over other types and should be recognizable to those familiar with other polymorphic object calculi, for example FGJ [63][1]. Permission variables are more interesting.

A permission variable, $\alpha$, can be introduced for the scope of an entire class, or a method. Each permission variable must be declared with an associated *quantification classifier*, $\kappa$, whose syntax is described in Figure 5.5. This classifier determines what a variable can be used for

[1]We have included traditional parametric polymorphism in order to make our examples more compelling. While we have left out more interesting features like F-bounded polymorphism, we believe that these features are orthogonal and can be added without any great difficulty.

$$
\begin{array}{rrcl}
\textit{quant. class.} & \kappa & ::= & \alpha \mid \mathsf{Asmp}(n,\kappa,T) \mid \omega \mid \Omega(\omega,\omega) \\
 & & \mid & \xi \mid \Xi(\xi) \mid \mathsf{Node}_T \\
\textit{fract. funct. type} & \omega & ::= & \mathsf{FF}(n,n,T) \mid \mathsf{UFF}(n,n,T) \\
\textit{fract. type} & \xi & ::= & \mathsf{Fract} \mid \mathsf{Decimal} \mid \mathbf{1} \mid \mathbf{0} \mid \mathsf{LessThan1} \mid \mathsf{GreaterThan0} \\
\textit{inst. elems.} & a & ::= & \alpha \mid A \mid \omega \mid \xi \mid k \mid g \\
\textit{permissions} & p & ::= & \mathsf{access}(r,n,g,k,A) \mid \mathsf{unpacked}(n,g,k,A) \\
\textit{facts} & q & ::= & t = \mathtt{true} \mid t = \mathtt{false} \\
\textit{assumptions} & A & ::= & \alpha \mid n \mid A_1 \otimes A_2 \\
\textit{fraction fct.} & g & ::= & \alpha \mid n \mapsto k \mid g/2 \mid g_1, g_2 \\
\textit{fractions} & k & ::= & \alpha \mid 1 \mid 0 \mid k/2 \\
\textit{predicates} & P & ::= & p \mid q \mid P_1 \otimes P_2 \mid \mathbf{1} \mid P_1 \,\&\, P_2 \mid \top \mid P_1 \oplus P_2 \mid \mathbf{0} \\
 & & \mid & \exists \alpha{:}\kappa.P \\
\textit{expr. types} & E & ::= & \exists x : T.P \\
\textit{fract. terms} & h & ::= & g \mid k \\
\textit{valid context} & \Gamma & ::= & \cdot \mid \Gamma, \textbf{\textit{CL}} \mid \Gamma, x{:}T \mid \Gamma, \beta \mid \Gamma, \alpha{:}\kappa \mid \Gamma, q \\
\textit{linear context} & \Delta & ::= & \cdot \mid \Delta, P
\end{array}
$$

*quantification variables* $\alpha$

Figure 5.5: Syntax II: Permissions, Abstraction and Checking

within its scope, and what sort of permission element can be instantiated for it. Those familiar with bounded parametric polymorphism should think of classifiers as being like type bounds for permission elements. The instantiating elements, $a$, are applied at the site of the method call or object instantiation expressions and become part of the class types, $C\langle\overline{T}\rangle[\overline{a}]$.

But what is the nature of the quantification classifiers? Recall that an access permission, $p$ in Figure 5.5, has the following form:

$$\mathsf{access}(r,n,g,k,A)$$

Our system allows each element, with the exception of the reference with which the permission is associated, to be abstracted. Therefore, depending on the classifier that is used, a newly introduced variable can stand for $n$, $g$, $k$, or $A$. The forms of the quantification classifier, $\kappa$, therefore are Node, a node (state or dimension) type, $\omega$, a fraction function type, $\xi$, a fraction type, and Asmp, an assumption type, respectively. Variables of type $\omega$ can only be instantiated with fraction functions, and variables of type $\xi$ can only be instantiated with fractions, etc. The fact that these newly introduced quantification variables can be used as elements of the access permission is reflected in the syntax, as $\alpha$ appears as a valid form of the syntactic categories $n$, $g$, $k$, and $A$.

The three other forms of quantification classifiers, $\alpha$, $\Omega$, and $\Xi$, are used to further abstract over the classifiers themselves, "one level up." They will be covered in a subsequent section. Thus far we have also neglected to discuss the various adornments of the quantification classifiers, such as $n$, $\kappa$ and $T$ in the classifier $\mathsf{Asmp}(n,\kappa,T)$. These adornments form an overall part of the bound on the quantification variable, and as we will show in the next section, are

necessary in order to ensure that access permissions that mention quantification variables remain well-formed.

### 5.3.2 Static Semantics for Permissions Abstraction and Application

Every time a programmer writes down a specification, which may consist of a number of access permissions, the static semantics of our language ensure that those permissions are well-formed. The system's well-formedness rules prevent certain programmer mistakes, such as the use of abstract states that have not been defined. These well-formedness rules motivate many of the features of our quantification classifiers. Let us consider the permission well-formedness rule presented in Chapter 3, which comes from Bierhoff [13]:

$$\textsc{Old WF-Perm}\quad \frac{\Gamma \vdash r : C \qquad C \vdash A \prec n \qquad \Gamma \vdash g : \overline{n} \mapsto \mathsf{Fract} \qquad \mathsf{nodes}(C) = \overline{n}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \mathsf{wf}}$$

This is to say that, in some type-checking context, a permission is well-formed if the reference has class type $C$, the assumption $A$ only mentions nodes below or equal to the guaranteed node $n$ in the state hierarchy of $C$, the fraction function $g$ maps a sequence of nodes $\overline{n}$ to fractions, those nodes include all of nodes of $C$ between alive and $n$, and $k$ is a fraction. Since at the time that quantification variables are introduced it is not known exactly which permission elements will be instantiated for them, it is the job of the quantification classifiers to ensure that a well-formed permission mentioning quantification variables will remain well-formed when those variables are instantiated.

Suppose we wanted to create a simple class that holds a field of parametrized type and with parametrized permission. Here is how we might declare such a class:

```
class OneField<β>[αn:Node_β, αg:FF(αn, alive, β), αk:Fract, αA:Asmp(αn, Fract, β)]
    extends Object <>[] {
  f : β in alive
  ...
  alive = access(f, αn, αg, αk, αA) // State invariant
  ...
}
```

Let us examine each classifier in turn. $\alpha_n$, an abstraction of a guaranteed node, is declared to have the classifier $\mathsf{Node}_\beta$. This classifier says that $\alpha_n$ must be instantiated with a node, and that node must be a node in the state hierarchy of type $\beta$. While we do not, as of yet, know what this type will be, $\alpha_n$ will be instantiated after the type variable $\beta$, at which point it will be clear whether or not the instantiated node is a node of the instantiated type. Next, $\alpha_g$ is classified as $\mathsf{FF}(\alpha_n, \mathsf{alive}, \beta)$. This tells us that $\alpha_g$ can only be instantiated with fraction functions, and those fraction functions must contain a fraction for every node in the state hierarchy of $\beta$ between $\alpha_n$ and alive, inclusive. Note how the bound of one quantification variable is dependent on other quantification variables. The classifier of $\alpha_k$, Fract says that it can only be instantiated with a fraction. Finally, the classifier for $\alpha_A$, $\mathsf{Asmp}(\alpha_n, \mathsf{Fract}, \beta)$ records that the variable can only be instantiated with assumptions (i.e., the syntactic form $A$). Furthermore, it stipulates that the instantiating assumption must be below $\alpha_n$ in the state hierarchy of type $\beta$ and, if the classifier

$$\textsc{Var}\ \frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \textsc{Subsm}\ \frac{\Gamma \vdash a : \kappa \qquad \Gamma \vdash \kappa \sqsubseteq \kappa'}{\Gamma \vdash a : \kappa'} \qquad \Gamma \vdash \omega : \Omega(\omega, \omega)$$

$$\Gamma \vdash \xi : \Xi(\xi) \qquad \Gamma \vdash 0 : \mathbf{0} \qquad \Gamma \vdash 1 : \mathbf{1}$$

$$\frac{\Gamma \vdash k : \mathsf{GreaterThan0}}{\Gamma \vdash k/2 : \mathsf{Decimal}} \qquad \textsc{Same Fract}\ \frac{\Gamma \vdash k : \kappa \qquad \kappa \sqsubseteq \mathsf{LessThan1}}{\Gamma \vdash k/2 : \kappa}$$

$$\frac{\Gamma \vdash k : \mathsf{Fract}}{\Gamma \vdash k/2 : \mathsf{Fract}} \qquad \textsc{FF-Div2}\ \frac{\Gamma \vdash g : \mathsf{FF}(n_1, n_2, T)}{\Gamma \vdash g/2 : \mathsf{FF}(n_1, n_2, T)} \qquad \frac{\Gamma \vdash g : \mathsf{UFF}(n_1, n_2, T)}{\Gamma \vdash g/2 : \mathsf{FF}(n_1, n_2, T)}$$

$$\frac{\Gamma \vdash k : \mathsf{Decimal} \qquad \Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n \mapsto k : \mathsf{FF}(n, n, T)} \qquad \frac{\Gamma \vdash k : 1 \qquad \Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n \mapsto k : \mathsf{UFF}(n, n, T)}$$

$$\frac{\Gamma \vdash g_1 : \mathsf{UFF}(n, n', T) \qquad \Gamma \vdash g_2 : \mathsf{FF}(n', n'', T)}{\Gamma \vdash g_1, g_2 : \mathsf{UFF}(n, n'', T)}$$

$$\frac{\Gamma \vdash g_1 : \mathsf{FF}(n, n', T) \qquad \Gamma \vdash g_2 : \mathsf{FF}(n', n'', T)}{\Gamma \vdash g_1, g_2 : \mathsf{FF}(n, n'', T)}$$

Figure 5.6: Classification of fractions and fraction functions.

Fract can classify fractions below one (which is trivially true!) the instantiating element for $\alpha_A$ must be equal to $\alpha_n$.

This last restriction deserves some mention. If a collection holds elements of share or pure permission kind, it must account for the fact that the state of these elements can be changed under the guaranteed node at any time. The assumption is therefore tied to the guarantee and can only be below the guarantee if the eventual instantiating fraction for $\alpha_k$ is one. (At the moment, $\alpha_A$ must trivially always be equal to $\alpha_n$, but after "classifier classifiers" are introduced, this will no longer be the case.)

The responsibility of ensuring that an instantiating permission element, $a$, satisfies the bound imposed on it by a quantification classifier, $\kappa$ falls on our type system. This is accomplished with the judgment, $\Gamma \vdash a : \kappa$, which says that under a valid typing context $\Gamma$, the instantiating element $a$ can be classified with $\kappa$. The rules for this judgment are shown in Figures 5.6 and 5.7.

Some discussion of these rules is in order. The VAR rule says that any quantification variable has the classifier that it was declared to have. The SUBSM rule says that any element $a$ with classifier $\kappa$ can be treated as being of classification $\kappa'$ if $\kappa$ is a sub-classifier of $\kappa'$. The next two rules say that classifiers themselves have classifiers, which we will motivate later. Every fraction form has a classifier, including the literals $1$ and $0$, whose classifiers are the literals themselves.

$$\textsc{Alive}\quad \Gamma \vdash \mathsf{alive} : \mathsf{Node}_T \qquad \textsc{Ground}\quad \frac{\Gamma \vdash n \ \mathsf{from}\ C}{\Gamma \vdash n : \mathsf{Node}_{C\langle\overline{\beta}\rangle[\overline{a}]}} \qquad \frac{\Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash n : \mathsf{Asmp}(n, \kappa, T)}$$

$$\frac{\Gamma \vdash n : \mathsf{Node}_T \qquad n \leq n'}{\Gamma \vdash n : \mathsf{Asmp}(n', \mathbf{1}, T)}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Asmp}(n', \mathbf{1}, T) \qquad \Gamma \vdash A_2 : \mathsf{Asmp}(n'', \mathbf{1}, T) \qquad \Gamma \vdash n' \leq n \qquad \Gamma \vdash n'' \leq n}{\Gamma \vdash A_1 \otimes A_2 : \mathsf{Asmp}(n, \mathbf{1}, T)}$$

Figure 5.7: Classification of nodes and assumptions.

Fraction functions can be classified as either FF, the classification of all fraction functions, or as UFF, the classification of *unique* fraction functions, that is fraction functions whose lowest node maps to the fraction 1. Rule ALIVE says that alive is a node for any type. Rule GROUND says that a node is defined in class $C$ at any instantiation if it is declared in class $C$. Finally, any node $n$ can be an assumption below or equal to node $n$ for any fraction classifier, but two assumptions can only be joined to form an assumption if both assumptions are below some common node $n$, and the classifier bound in Asmp is $\mathbf{1}$.

Now that we have seen the variety of classifiers available in our type system and how each permission element is classified, let us present the new well-formedness rule for permissions, which updates OLD WF-PERM presented earlier in this section:

WF-PERM

$$\frac{\Gamma \vdash g : \mathsf{FF}(n, \mathsf{alive}, T) \qquad \begin{array}{l}\Gamma \vdash r : T \\ \Gamma \vdash k : \kappa\end{array} \qquad \begin{array}{l}\Gamma \vdash n : \mathsf{Node}_T \\ \Gamma \vdash \kappa \sqsubseteq \mathsf{Fract}\end{array} \qquad \Gamma \vdash A : \mathsf{Asmp}(n, \kappa, T)}{\Gamma \vdash \mathsf{access}(r, n, g, k, A)\ \mathsf{wf}}$$

Thanks to our changes, the classifiers of each element of the permission succinctly express the restrictions on each element. Note that the classifier of $k$, $\kappa$ is the same $\kappa$ mentioned in $A$'s classifier. This restriction, coupled with the assumption classification rules in Figure 5.7, ensure that $n = A$ for any polymorphic permission with a fraction $k$ less than one. Using WF-PERM, our type system would find that the state invariant for the alive state in the `OneField` class is indeed well-formed:

$$\beta,\ f{:}\beta,\ \alpha_n{:}\mathsf{Node}_\beta,\ \alpha_g{:}\mathsf{FF}(\alpha_n, \mathsf{alive}, \beta),\ \alpha_k{:}\mathsf{Fract},\ \alpha_A{:}\mathsf{Asmp}(\alpha_n, \mathsf{Fract}, \beta) \vdash \mathsf{access}(f, \alpha_n, \alpha_g, \alpha_k, \alpha_A)\ \mathsf{wf}$$

The quantification classifiers also form a number of interesting sub-classification relationships. Sub-classification allows programmers to write specifications that are quite expressive, in a way that is analogous to Java's F-bounded polymorphism. Sub-classification is established with the judgment $\Gamma \vdash \kappa \sqsubseteq \kappa$. The rules for this judgment are presented in Figure 5.8.

The main points of interest are the relationships between fraction classifiers, and the relationships between fraction function classifiers. Fraction classifiers form a hierarchy from Fract, the

$$\textsc{Reflexive}\quad \Gamma \vdash \kappa \sqsubseteq \kappa \qquad \textsc{Transitive}\quad \frac{\Gamma \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \vdash \kappa \sqsubseteq \kappa''} \qquad \Gamma \vdash 1 \sqsubseteq \mathsf{GreaterThan0}$$

$$\Gamma \vdash \mathsf{Decimal} \sqsubseteq \mathsf{GreaterThan0} \qquad \Gamma \vdash \mathsf{GreaterThan0} \sqsubseteq \mathsf{Fract}$$

$$\Gamma \vdash 0 \sqsubseteq \mathsf{LessThan1} \qquad \Gamma \vdash \mathsf{Decimal} \sqsubseteq \mathsf{LessThan1}$$

$$\Gamma \vdash \mathsf{LessThan1} \sqsubseteq \mathsf{Fract} \qquad \Gamma \vdash \mathsf{UFF}(n_1, n_2, T) \sqsubseteq \mathsf{FF}(n_1, n_2, T)$$

$$\frac{\Gamma \vdash n \leq n' \text{ in } T \qquad \Gamma \vdash \kappa \sqsubseteq \kappa'}{\Gamma \vdash \mathsf{Asmp}(n, \kappa, T) \sqsubseteq \mathsf{Asmp}(n', \kappa', T)} \qquad \frac{\Gamma \vdash \omega_1' \sqsubseteq \omega_1 \qquad \Gamma \vdash \omega_2 \sqsubseteq \omega_2'}{\Gamma \vdash \Omega(\omega_1, \omega_2) \sqsubseteq \Omega(\omega_1', \omega_2')}$$

$$\frac{\Gamma \vdash \xi \sqsubseteq \xi'}{\Gamma \vdash \Xi(\xi) \sqsubseteq \Xi(\xi')} \qquad \textsc{FF Upper-Bound}\quad \frac{\Gamma \vdash \alpha : \Omega(\_, \omega)}{\Gamma \vdash \alpha \sqsubseteq \omega} \qquad \textsc{FF Lower-Bound}\quad \frac{\Gamma \vdash \alpha : \Omega(\omega, \_)}{\Gamma \vdash \omega \sqsubseteq \alpha}$$

$$\textsc{Fract Upper-Bound}\quad \frac{\Gamma \vdash \alpha : \Xi(\xi)}{\Gamma \vdash \alpha \sqsubseteq \xi}$$

Figure 5.8: Sub-classification rules

classifier of every fraction, to **0**, **1**, and Decimal, the classifiers for $0$, $1$, and fractions between $0$ and $1$, respectively. LessThan1 and GreaterThan0 have the obvious locations in this hierarchy. Fraction functions can be classified by FF, or its sub-classifier UFF. Fraction functions classified by UFF have their lowest node mapped to $1$, and are the fraction functions used for unique permissions.

### 5.3.3 Abstracting Over Quantification Classifiers

While the ability to abstract over fractions and fraction functions is useful, it is not quite as flexible as we would like. Consider the following scenario: We would like to take our stack, presented back in Section 5.1, and specify it generically over the permission kind of the elements it holds, but where every each element must be of the same kind. We will only concentrate on the class quantification variables and the `push` method, since this will be enough to motivate higher quantification. Consider the following specification of `Stack`:

```
class Stack<β>[αn:Node_β, αg:FF(αn, alive, β), αk:Fract, αA:Asmp(αn, Fract, β)]
  extends Object <>[] { ...
  boolean push(T i) : unique(this) ⊗ access(i, αn, αg, αk, αA) ⊸ unique(this)
  ... }
```

Now, suppose that at a particular instantiation site, we would like to use this stack, and we would like it instantiated as a stack of shared permissions to files that are guaranteed to be open. What instantiations should we use? Unfortunately, we are required to choose definite values for the fraction $\alpha_k$, and the fraction function, $\alpha_g$. Let us assume that we instantiate the stack as follows; $\texttt{Stack}\langle\texttt{File}\rangle[\mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open}]$. This means that in an environment where the permission $\mathsf{access}(r_1, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open})$, a share permission, is available for $r_1$, the call $\texttt{push}(r_1)$ is legal. Unfortunately, if we have another share permission with different fraction values, say $\mathsf{access}(r_2, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{4}, \mathsf{Open} \mapsto \frac{1}{4}\}, \frac{1}{4}, \mathsf{Open})$, the call $\texttt{push}(r_2)$ is *not* legal, because the pre-condition for the push method when instantiated is $\mathsf{unique}(\texttt{stack}) \otimes \mathsf{access}(r_1, \mathsf{Open}, \{\mathsf{alive} \mapsto \frac{1}{2}, \mathsf{Open} \mapsto \frac{1}{2}\}, \frac{1}{2}, \mathsf{Open})$. This requires the *exact* same fraction and fraction function values.

To accomplish our original goal of instantiating a stack that can hold share permissions at any fraction, we need more power in the specification language. We need the ability to quantify over the classifiers themselves. Fortunately, the quantification classifiers $\Omega$ and $\Xi$ let us do exactly that. $\Omega$ is the classifier of all fraction function classifiers. It stores an upper bound and a lower bound of the classifiers that can legally be used to instantiate it. $\Xi$ is the classifier of fraction classifiers. It stores an upper bound of the classifiers that can legally be used to instantiate it. (Why no lower bound? It was not found to be useful for any of our examples. Adding it would be fairly straightforward.) By abstracting over these "classifier classifiers," we can specify that certain fractions and fraction functions must be similar but not identical.

With $\Omega$ and $\Xi$ at our disposal, we can correctly specify the Stack class:

```
class Stack<β>[αn:Node_β, αω:Ω(UFF(αn, alive, β), FF(αn, alive, β)),
 αξ:Ξ(Fract), αA:Asmp(αn, αξ, β)] extends Object <>[] {
  ...
  boolean push(T i) :
     unique(this) ⊗ (∃αg:αω.∃αk:αξ.access(i, αn, αg, αk, αA)) ⊸ unique(this)
  ...
}
```

With a stack instantiated as, $\texttt{Stack}\langle\mathsf{File}\rangle[\mathsf{Open}, \mathsf{FF}(\mathsf{Open}, \mathsf{alive}, \texttt{File}), \mathsf{Decimal}, \mathsf{Open}]$, we can call the push method and pass share permissions of any fractional value. This is in part thanks to the existential quantification that has been added to the push method's specification. When instantiated, it can accept any fraction as long as that fraction is classified by Decimal, and any fraction function, provided it is classified by FF.

### 5.3.4 Quantifying Over Symmetric Permission Kinds

Up until this point, we have used Stack as a running example. One of the notable features of stack is that it can hold permissions of any kind. This is largely due to its implementation. A programmer can push an object, and the stack will capture some permission associated with that object. Later on, when the pop method is called, the entire permission to the returned element is forfeited by the stack. This means that no matter what permission kind the stack holds, we can count on getting it back later in the execution.

However, some data structures do not provide this feature, and yet could still reasonably support multiple permission kinds. The polymorphic type system we have presented here allows

```
1  class LinkedList<T> {
2    class Node {  T item; Node next;
3
4      T get(int i, int cur) {
5        if( i == cur ) return item;
6        else return next == null ? null :
7             next.get(i, cur + 1);
8      }
9    }
10
11   int size = 0; Node first = null;
12
13   int size() {...}  void add(T item) {...}
14
15   T get(int i) {
16     if( first == null ) return null;
17     else return first.get(i,0);
18   }
19 }
```

Figure 5.9: A linked list that provides random access to its elements.

us to precisely specify the behavior of these classes. Consider the mutable linked list class shown in Figure 5.9. It, like many of the collection classes in the Java standard library, provides random access to its elements. If we would like to use this list in a larger program, we must ask what kind of permission we can get back from the `get` method, especially in light of multiple requests for the same element:

```
Object o_1 = list.get(0);
Object o_2 = list.get(0);
```

Does the second call return the same permission? Does it return no permission? Does it generate an error? There are multiple ways we might want our list to behave. One observation is that this linked list can hold elements of any permission kind that can be split indefinitely to produce the same permission. We call such permissions "symmetric," and both the share and pure permissions have this property (along with the immutable permission, which is not part of our formal treatment). Using classifier bounds, our type system allows us to specify `LinkedList` in such a way that it can be used for share and pure but not full or unique.

Here is how we might specify the `LinkedList` class: First, we will introduce bounded quantifiers at the class level:

```
class LinkedList<β>[αn:Node_β,  αω:Ω(FF(αn, alive, β), FF(αn, alive, β)),
 αξ:Ξ(LessThan1),  αA:Asmp(αn, αξ, β)] {
...
}
```

Here note that the fraction classifier $\alpha_\xi$ is bounded so that it can never classify any fraction whose value is 1 (which would be necessary for a unique or full permission). The fraction function classifier $\alpha_\omega$ is bounded from below by FF, which means that it can never be used to classify a unique fraction function (the fraction function that would be used in a unique permission).

The effect of these bounds are two-fold. First, they prevent unique and full permissions from ever being used to instantiate the linked list. This generally means that a unique or full permission cannot be returned as a result of calling the `get` method, although because of splitting these permissions could still be used to satisfy the pre-condition of the `add` method. Secondly, these bounds give the analysis enough information to know internally that fractions classified by $\alpha_\xi$ and fraction functions classified by $\alpha_\omega$ can be split and still result in a fraction of the same classification. Rules SAME FRACT and FF-DIV2 in Figure 5.6 make this possible.

To better illustrate this idea, let us attempt to verify an implementation of the `get` method of the `Node` class, beginning on line 4 of Figure 5.9. Here is a specification along with an implementation, assuming the quantified variables introduced in the previous listing are in scope:

```
class Node {
  alive = unique(next) ⊗ (∃αg:αω.∃αk:αξ.access(result, αn, αg, αk, αA))
  ...
  β get(int i, int cur) :
  unique(this) ⊸ (∃αg:αω.∃αk:αξ.access(result, αn, αg, αk, αA)) ⊗ unique(this) {
    if( i == cur,
        unpack(alive,1,alive) in
        let r = item in
        pack alive to alive in r,
      let n = next in
      if( n == null, null, n.get(i,cur+1) )
  }
  ...
}
```

Verifying the `get` method requires proving the permission

$$\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$$

twice, once to satisfy the post-condition, and once to enable the receiver to be packed to the alive state. While splitting rules essentially always allow an access permission to be split in two, it is the bounds on the classification variables that ensure the fraction and fraction function are still classified by $\alpha_\xi$ and $\alpha_\omega$ after being divided by two. In other words, rules SAME FRACT and FF-DIV2 allow the following verification condition to succeed:

$$(\alpha_\omega{:}\Omega(\ldots),\ \alpha_g{:}\alpha_\omega,\ \alpha_\xi{:}\Xi(\ldots),\ \alpha_k{:}\alpha_\xi); \mathsf{access}(\mathit{result}, \alpha_n, \alpha_g/2, \alpha_k/2, \alpha_A) \vdash \exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(\mathit{result}, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$$

### 5.3.5 Typing Rules

With a few exceptions, the the type-checking rules for expressions in our language are extensions of the rules presented in Chapter 3. Therefore, Figure 5.10 presents only the rules that have changed due to polymorphism. The main typing judgment is $\Gamma; \Delta \vdash_C e : E$, which means, in the context of some valid facts $\Gamma$, some linear facts $\Delta$, and within the context of class $C$, the expression $e$ has type $E$.

Rule P-NEW checks an instantiation expression. After checking that the instantiated type is well-formed, the init function takes an instantiated class type and returns the types of its fields, the

P-NEW

$$\frac{\Gamma \vdash C\langle\overline{T_0}\rangle[\overline{a}]\ \mathsf{wf} \qquad \Gamma \vdash \mathsf{init}(C\langle\overline{T_0}\rangle[\overline{a}]) = \langle\overline{f:T}, \overline{\alpha:\kappa}, P, A\rangle \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma;\Delta \vdash [\overline{t/f}]P}{\Gamma;\Delta \vdash_C \ \mathtt{new}\ C\langle\overline{T_0}\rangle[\overline{a}](\overline{t}) : \exists x{:}C\langle\overline{T_0}\rangle.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, 1, A)}$$

P-ASSIGN

$$\frac{\Gamma;\Delta;\Psi \vdash t : \exists x : T_i.P \qquad p = \mathsf{unpacked}(n, g, k, A) \qquad \alpha \notin A \qquad \Gamma;\Delta' \vdash [f_i/x']P' \otimes p \qquad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad n_i \leq n \qquad \Gamma \vdash k : \mathsf{GreaterThan0}}{\Gamma;(\Delta,\Delta');\Psi \vdash^C f_i := t : \exists x' : T_i.[f_i/x]P \otimes P' \otimes p}$$

P-PACK

$$\frac{\Gamma;\Delta \vdash_C \mathsf{inv}_C(n, g, k, A) \otimes \mathsf{unpacked}(n, g, k, A') \qquad n \neq \alpha \qquad \alpha \notin A \vee A = A' \qquad \Gamma \vdash k : \kappa \qquad \Gamma \vdash 0 \sqsubseteq \kappa \textit{ implies } A = A' \qquad \Gamma;(\Delta', \mathsf{access}(\textit{this}_{\mathsf{fr}}, n, g, k, A)) \vdash_C e : E \qquad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \qquad \textit{Fields do not occur in } \Delta'}{\Gamma;(\Delta,\Delta') \vdash_C \mathtt{pack}\ n\ \mathtt{to}\ A\ \mathtt{in}\ e : E}$$

P-CALL

$$\frac{\textit{no unpacked perms in } \Delta \qquad \Gamma \vdash t_0 : C\langle\overline{T_0}\rangle[\overline{a_0}] \qquad \Gamma \vdash \mathsf{sargs}(m, C\langle\overline{T_0}\rangle[\overline{a_0}]) = (\overline{\alpha:\kappa}) \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma \vdash \mathsf{mtype}(m[\overline{a}], C\langle\overline{T_0}\rangle[\overline{a_0}]) = (\overline{x:T}, P \multimap E) \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma;\Delta \vdash [t_0/\mathtt{this}][t_0/\textit{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P}{\Gamma;\Delta \vdash_C t_0.m[\overline{a}](\overline{t}) : [t_0/\mathtt{this}][\overline{t}/\overline{x}]E}$$

P-SUPER

$$\frac{\textit{no unpacked perms in } \Delta \qquad \Gamma \vdash \mathtt{this} : C\langle\overline{T_t}\rangle[\overline{a_t}] \qquad \Gamma \vdash \mathsf{stype}(C\langle\overline{T_t}\rangle[\overline{a_t}]) = C'\langle\overline{T_s}\rangle[\overline{a_s}] \qquad \Gamma \vdash \mathsf{sargs}(m, C'\langle\overline{T_s}\rangle[\overline{a_s}]) = (\overline{\alpha:\kappa}) \qquad \Gamma \vdash \overline{a:\kappa} \qquad \Gamma \vdash \mathsf{mtype}(m[\overline{a}], C'\langle\overline{T_s}\rangle[\overline{a_s}]) = (\overline{x:T}, P \multimap E) \qquad \Gamma \vdash \overline{t:T} \qquad \Gamma;\Delta \vdash [\textit{super}/\textit{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P}{\Gamma;\Delta \vdash_C \ \textit{super}.m[\overline{a}](\overline{t}) : [\textit{super}/\textit{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}$$

Figure 5.10: Expression typing rules modified due to polymorphism

classifications of the polymorphic variables, the initial object state $A$ and the state invariants for that state $P$. Both the types of the fields and the initial state invariant are returned in terms of the *instantiating* types and permission elements, as the definition of the init function in Figure 5.11 explains. The rule then checks that the instantiating elements $\overline{a}$ are actually classified by $\overline{\kappa}$ and then uses the current linear context to prove the required permissions $P$, but for the arguments that are passed to the constructor, rather than the fields.

Interestingly, the rules for unpacking do not change. The receiver can essentially be unpacked at any time. As long as some permission is available to the receiver, it does not matter if that permission is generic. What *does* change somewhat is the invariant look-up function, $\mathsf{inv}_C$. This function determines what permission is actually produced when an object is unpacked, and is revised as seen in Figure 5.13. The changes make it so that the empty permission, **1**, will be

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T'}\rangle[\overline{a'}]\{\ldots\ \overline{f:T\ \texttt{in}\ n}\ \texttt{initially}\langle P, s_1 \otimes \ldots \otimes s_n\rangle\ldots\} \in \Gamma \quad \Gamma \vdash \mathsf{init}(C'\langle\overline{T'}\rangle[\overline{a'}]) = (\overline{f'':T''}, \overline{\alpha':\kappa'}, P', A') \quad \Gamma; (P, \mathsf{access}(\textit{super}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, A')) \vdash \mathsf{inv}_C(\mathsf{alive}, A) \otimes \top}{\Gamma \vdash \mathsf{init}(C\langle\overline{T}\rangle[\overline{a}]) = ([\overline{T}/\overline{\beta}][\overline{a}/\overline{\alpha}](\overline{f:T}, \overline{f'':T''}), [\overline{a}/\overline{\alpha}](P \otimes P'), A)}$$

$$\Gamma \vdash \mathsf{init}(\texttt{Object}\langle\rangle[]) = (\cdot, \cdot, 1, \mathsf{alive})$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\ \overline{M}\ldots\} \quad T\ m[\overline{\alpha_m:\kappa_m}](\overline{T\ x}) : P \multimap E = e \in \overline{M} \qquad \overline{\kappa'_m} = [\overline{a}/\overline{\alpha}]\overline{\kappa_m}}{\Gamma \vdash \mathsf{sargs}(m, C\langle\overline{T}\rangle[\overline{a}]) = (\overline{\alpha_m:\kappa'_m})}$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\ \overline{M}\ldots\} \quad T\ m[\overline{\alpha_m:\kappa_m}](\overline{T\ x}) : P \multimap E = e \in \overline{M} \quad \overline{T'} = [\overline{T_c}/\overline{\beta}]\overline{T} \qquad P' = ([\overline{a}/\overline{\alpha_m}]([\overline{a_c}/\overline{\alpha}]P)) \qquad E' = [\overline{T_c}/\overline{\beta}]([\overline{a}/\overline{\alpha_m}]([\overline{a_c}/\overline{\alpha}]E))}{\Gamma \vdash \mathsf{mtype}(m[\overline{a}], C\langle\overline{T_c}\rangle[\overline{a_c}]) = (\overline{x:T'}, P' \multimap E')}$$

$$\frac{\texttt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \texttt{extends}\ C'\langle\overline{T_s}\rangle[\overline{a_s}]\{\ldots\} \in \Gamma \qquad \overline{T'} = [\overline{T}/\overline{\beta}]T_s \qquad \overline{a'} = [\overline{a}/\overline{\alpha}]a_s}{\Gamma \vdash \mathsf{stype}(C\langle\overline{T}\rangle[\overline{a}]) = C'\langle\overline{T'}\rangle[\overline{a'}]}$$

Figure 5.11: Various utility judgments used by type-checking and well-formedness rules.

returned whenever an object is unpacked from an assumed state or a guaranteed state that is a variable, i.e., $\alpha_A$ or $\alpha_n$. A useful state invariant will only be returned if the unpacked state is a real state, e.g., "Open." The rule for field assignment, P-ASSIGN also needs to change. The main difference is the restriction on the unpacked permission and its assumption $A$. An unpacked receiver can only be assigned to if the unpacked state is completely ground, here signified by the restriction, $\alpha \notin A$. P-PACK is modified with a similar restriction, that the state to which an object is packed must either be identical to the unpacked state or a ground state. These two rules work in harmony to ensure that objects of generic permission can be unpacked only for reading purposes. Their fields cannot be reassigned, which might allow state invariants to be violated by a client packing to an unground state $\alpha_A$ with a trivially satisfiable invariant **1**.

The rule P-CALL checks a method call site. The receiver is checked to ensure that it has some kind of class type. The sargs function, defined in Figure 5.11, looks up the classifiers of the static method parameters, and then the permission arguments, $\overline{a}$, are checked to ensure they have the same classifiers. Additionally, the method arguments are checked to ensure that they have the same types as the method parameters. Note that the mtype function (Figure 5.11) takes into account the static arguments of $t_0$'s type, $C\langle\overline{T_0}\rangle[\overline{a_0}]$. Finally, the linear context is used to prove the method pre-condition, after all of the appropriate substitutions are made.

The rule for type-checking calls of superclass methods, P-SUPER, works very much in the same way. Our type system uses a "frames" methodology [31] for ensuring soundness in the face of subclassing, here evident in the appearance of the $\textit{this}_{\mathsf{fr}}$ and *super* references. This

P-CLASS

$$\frac{\mathsf{ftypes}(\overline{F}) = \overline{f:T} \qquad \Gamma' = \Gamma, \overline{\beta}, \overline{f:T}, \overline{a:\kappa} \qquad \Gamma' \vdash \overline{\kappa\ \mathsf{wf}} \qquad \Gamma' \vdash C'\langle\overline{T}\rangle[\overline{a}]\ \mathsf{wf} \qquad \Gamma' \vdash \overline{F}\ \mathsf{ok\ in}\ C \qquad \Gamma', \mathtt{this} : C\langle\overline{\beta}\rangle[\overline{\alpha}] \vdash \overline{M}\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha}] \qquad \Gamma' \vdash \overline{N}\ \mathsf{ok} \qquad \Gamma' \vdash I\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha}] \qquad \Gamma' \vdash \overline{R}\ \mathsf{ok\ in}\ C \qquad \overline{M}\textit{overrides all methods with this}_{\mathsf{fr}}\textit{ perm in C'}}{\Gamma \vdash \mathtt{class}\ C\langle\overline{\beta}\rangle[\overline{\alpha:\kappa}]\ \mathtt{extends}\ C'\langle\overline{T}\rangle[\overline{a}]\{\overline{F}\ \overline{R}\ I\ \overline{N}\ \overline{M}\}\ \mathsf{ok}}$$

P-METHOD

$$\frac{\Gamma' = \Gamma, \overline{\alpha:\kappa}, \overline{x:T} \qquad \Gamma' \vdash \overline{\kappa\ \mathsf{wf}} \qquad \Gamma' \vdash \overline{T\ \mathsf{wf}} \qquad \Gamma' \vdash P\ \mathsf{wf} \qquad \Gamma', \textit{result} : T_r \vdash P_r\ \mathsf{wf} \qquad \Gamma' \vdash \mathsf{override}(m, C\langle\overline{\beta}\rangle[\overline{\alpha_c}], \overline{x:T}, P \multimap \exists \textit{result} : T_r.P_r) \qquad \Gamma'; P \vdash_C e : \exists \textit{result} : T_r.P_r \otimes \top}{\Gamma \vdash m[\overline{\alpha:\kappa}](\overline{T\ x}) : P \multimap \exists \textit{result} : T_r.P_r = e\ \mathsf{ok\ in}\ C\langle\overline{\beta}\rangle[\overline{\alpha_c}]}$$

Figure 5.12: Well-formedness rules for the entire program.

rule, which does not appear in Chapter 3, is due to Bierhoff and Aldrich [15]. Their system allowed three types of receiver permissions, `this`, a virtual permission which could be used to dynamically dispatch, $\textit{this}_{\mathsf{fr}}$, a frame permission that could be used to unpack an object at one level of the subtype hierarchy, and *super*, a permission to the supertype from the point of view of a subtype. This powerful system allows for each level of the subtype hierarchy for a given object to inhabit a separate abstract state. We include these features here in order to allow our polymorphic extension to be applied to their system.

$$\frac{\Gamma \vdash k : \mathsf{GreaterThan0}}{\Gamma \vdash \mathsf{inv}_C(n, g, k, A) = \mathsf{inv}_C(n, A) \otimes \mathsf{purify}(\mathsf{above}_C(n))}$$

$$\frac{\Gamma \vdash k : \mathsf{Fract}}{\Gamma \vdash \mathsf{inv}_C(n, g, 0, A) = \mathsf{purify}(\mathsf{inv}_C(n, A) \otimes \mathsf{above}_C(n))} \qquad \frac{\mathtt{class}\ C\ \{\ldots n = P \ldots\} \in \overline{CL}}{\Gamma \vdash \mathsf{pred}_C(n) = P}$$

$$\frac{\Gamma \vdash n : \mathsf{Node}_T}{\Gamma \vdash \mathsf{pred}_C(n) = \mathbf{1}} \qquad \Gamma \vdash \mathsf{inv}_C(n) = \mathbf{1} \Rightarrow n$$

$$\frac{\Gamma \vdash \mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \qquad \Gamma \vdash \mathsf{pred}_C(n_i, n) = P_i' \qquad n_1 \oplus n_2 \ll n \qquad (i = 1, 2)}{\Gamma \vdash \mathsf{inv}_C(A_1 \otimes A_2) = P_1 \otimes P_1' \otimes P_2 \otimes P_2' \Rightarrow n}$$

$$\frac{\Gamma \vdash A : \mathsf{Asmp}(n, \kappa, T)}{\Gamma \vdash \mathsf{inv}_C(A) = \mathbf{1}}$$

Figure 5.13: The revised invariant look-up rules. All other rules are as originally presented.

Beyond the expression typing rules, there are also a number of rules for ensuring that an entire program is well-formed. These are given in Figure 5.12. Rule P-CLASS checks that a

class is well-formed by adding all of the fields, type variables and quantification variables to the valid context. Every declared quantification classifier is checked to ensure that it is well-formed. It then checks that the field, state, method, constructor and state invariant declarations are well-formed. Rule P-METHOD checks that a method's body correctly implements its specification. First, the quantification classifiers and argument types are checked for well-formedness. An augmented context is used to check that the pre- and post-conditions are well-formed. The override judgment checks that the method's specification is behaviorally compatible with any methods it overrides. Finally, given the permissions specified in the post-condition, the method body is type-checked to ensure that it correctly satisfies its post-condition.

$$\textsc{Var-}\Omega\ \frac{\Gamma \vdash \alpha : \Omega(\omega_1, \omega_2)}{\Gamma \vdash \alpha\ \mathsf{wf}} \qquad \textsc{Var-}\Xi\ \frac{\Gamma \vdash \alpha : \Xi(\xi)}{\Gamma \vdash \alpha\ \mathsf{wf}} \qquad \frac{\Gamma \vdash n_1 \leq n_2\ \mathsf{in}\ T \qquad \Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{FF}(n_1, n_2, T)\ \mathsf{wf}}$$

$$\frac{\Gamma \vdash n_1 \leq n_2\ \mathsf{in}\ T \qquad \Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{UFF}(n_1, n_2, T)\ \mathsf{wf}} \qquad \textsc{FF Type}\ \frac{\Gamma \vdash \omega_1\ \mathsf{wf} \qquad \Gamma \vdash \omega_2\ \mathsf{wf} \qquad \Gamma \vdash \omega_1 \sqsubseteq \omega_2}{\Gamma \vdash \Omega(\omega_1, \omega_2)\ \mathsf{wf}}$$

$$\textsc{Node}\ \frac{\Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{Node}_T\ \mathsf{wf}} \qquad \textsc{Assumption}\ \frac{\Gamma \vdash n : \mathsf{Node}_T \qquad \Gamma \vdash \kappa\ \mathsf{wf} \qquad \Gamma \vdash \kappa \sqsubseteq \mathsf{Fract}\Gamma \vdash T\ \mathsf{wf}}{\Gamma \vdash \mathsf{Asmp}(n, \kappa, T)\ \mathsf{wf}}$$

$$\textsc{Fracts}\ \Gamma \vdash \xi\ \mathsf{wf} \qquad \textsc{Fract Type}\ \Gamma \vdash \Xi(\xi)\ \mathsf{wf}$$

Figure 5.14: Rules for checking the well-formedness of quantification classifiers.

$$\textsc{F-Split}\ \otimes\ \frac{\begin{array}{c} n_1 \# n_2 \qquad A_1 \prec n_1 \leq n \qquad A_2 \prec n_2 \leq n \\ \Gamma \vdash k : 1 \qquad p_i = \mathsf{access}(r, n_i, \{g/2, \mathsf{nodez}(n_i, n) \mapsto 1/2, n_i \mapsto 1\}, k, A_i) \end{array}}{\Gamma \vdash \mathsf{access}(r, n, g, k, A_1 \otimes A_2) \Rrightarrow p_1 \otimes p_2}$$

$$\textsc{F-Down}\ \frac{A \prec n' \leq n \qquad \Gamma \vdash k : 1}{\Gamma \vdash \mathsf{access}(r, n, g, k, A) \Rrightarrow \mathsf{access}(r, n', \{g, \mathsf{nodes}(n', n) \mapsto 1\}, k, A)}$$

Figure 5.15: Splitting rules, revised to account for polymorphic permissions. The remainder of the splitting and joining rules are unchanged.

Interestingly, the rules for splitting and joining do not need to change very much from those first presented in Figure 3.18 of Chapter 3. In fact, only two splitting rules needed to be revised.

Those rules are shown in Figure 5.15. The original splitting and joining rules were very much syntax-driven, and with respect to permissions, the syntax has not changed so much as it has been augmented. Therefore all of the original rules still worked, we simply chose to modify two to allow for even more expressiveness. In the two modified rules, F-SPLIT $\otimes$ and F-DOWN, full permissions are required, and the rules show that they can be used to create either one of two new permissions of a lower guarantee. Our only change was to simply say that, to qualify as a full permission, a permission's below fraction $k$ need only *classify* as $1$, rather than actually being $1$. The rest of the rules are the same, with the small caveat that they should be checked under a valid context $\Gamma$.

### 5.3.6 Concurrency

In this chapter we have chosen to focus on parametric polymorphism for a single-threaded protocol analysis, specifically an extension to the system presented by Bierhoff and Aldrich [15]. Still, it is worth asking what would be required in order to make such a system integrate with the rules presented in Chapter 3, which are sound in the face of concurrency. The short answer is, not much.

Our standard approach consists of two rules which restore soundness in the face of concurrent access. One rule requiring us to downgrade all permissions that might be concurrently modified to their guaranteed state, unless a lock is held. The second rule requires a lock to be held in order to unpack an object of full, share or pure permission. In the polymorphic setting, unknown permissions can be declared, so how do we know statically whether or not they must be accessed within a lock?

Addressing the first rule in a polymorphic context actually requires us to do nothing. In the places where polymorphic permission can actually be declared by programmers, that is on methods and in state invariants, those permissions have to be well-formed. As part of a permission's being well-formed, the assumption must be equal to the guaranteed state unless the below fraction, $k$, can only be instantiated with the value $1$. Another way to put this is, we can be sure that the assumption state matches the guaranteed state unless the eventual instantiating permission must be a unique or a full, and those are exactly the permissions that do not need to be forgotten in an concurrent setting.

The restrictions on unpacking are similarly straightforward. In a concurrent setting, we would want to guarantee that when unpacking an object of polymorphic permission that might be instantiated with full, share or purepermission, the right lock is held. Therefore, the concurrent unpack rule, (P-UNPACK from Figure 3.23) would replace its restriction on the fraction function $g$ with the requirement that it be a unique fraction function, UFF. In all other cases, P-UNPACK-SYNC would apply, requiring that the lock be held. The Sync-or-Swim tool has been enhanced to support parametric polymorphism, and implements these required modifications.

## 5.4 Syntactic Sugar

Up until this point we have presented our extension as if programmers would be writing out the full specifications. This system is quite flexible and expressive. However, given the syntactic

complexity of some of the quantification bounds, for example $[\alpha_n$:Node$_\beta$, $\alpha_\omega$:$\Omega$(FF($\alpha_n$,alive, $\beta$), FF($\alpha_n$, alive, $\beta$)), $\alpha_\xi$:$\Xi$(LessThan1), $\alpha_A$:Asmp($\alpha_n, \alpha_\xi, \beta$)] from our linked list example, we would really like to simplify things a bit! In this section we will introduce syntactic sugar that greatly simplifies our system of polymorphic access permissions while still retaining most of the expressiveness.

In order to simplify our system, we will introduce polymorphic variables that stand for entire access permissions, rather than for each permission element. These variables, when introduced, will be declared with one of three types of bounds:

**Exact** This variable bound introduces a permission that refers to a specific fractional quantity. Every time it is used, the instantiated permission will be required to be exactly the same.

**Similar** This variable bound introduces what is essentially a family of permissions each of the same permission kind. Every time this permission variable is used, instantiations are required to be of the same kind, but not necessarily the same fraction.

**Symmetric** This variable bound introduces a permission variable that is identical to 'Similar' in every way, and additionally can be divided an infinite number of times. Therefore, it can only be instantiated with permissions of kind pure and share (and immutable in our implementation).

Using these simplified bounds, the linked list class presented in the previous section could be written in the following manner:

```
class LinkedList<β>[p : symmetric(β)] {
  class Node {
    β item; Node next;
    alive = unique(next) ⊗ p(item)

    β get(int i, int cur):unique(this) ⊸ unique(this) ⊗ p(result)
  }

  alive = unique(first)
  ...
  void add(β item) : unique(this) ⊗ p(item) ⊸ unique(this)

  β get(int i) : unique(this) ⊸ unique(this) ⊗ p(result)
}
```

The permission variable $p$ stands for a permission that can be divided any number of times but will still result in a permission of the same kind. Specifically, each time $p$ is mentioned, it may refer to different fractions in the below fraction and the fraction function. Note that the bound of $p$ must still declare the type $\beta$ with which its permissions will be associated.

These new permission variables are truly syntactic sugar. They can be defined in terms of our lower level quantification variables. For each of the three types of bounds for permission variables, there is a different way to translate its declaration and its use. The table in Figure 5.16 summarizes the transformation from syntactic sugar to the formal language.

Of particular note is the translation of the use of a similar or symmetric permission variable. Each use is translated into an access permission that existentially quantifies the fraction and fraction functions. The classifiers of these existentially quantified variables are the classifiers introduced when the permission variable itself was declared. Additionally, the symmetric

| Declaration and Use | |
|---|---|
| **Sugar** | **Rewrite** |
| $p$ : exact$(T)$ | $\alpha_\xi : \Xi(\mathsf{Fract}), \alpha_n : \mathsf{Node}_T, \alpha_g : \mathsf{FF}(\alpha_n, \mathsf{alive}, T), \alpha_k : \alpha_\xi,$ $\alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$ |
| $p(r)$ | $\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$ |
| $p$ : similar$(T)$ | $\alpha_n : \mathsf{Node}_T, \alpha_\omega : \Omega(\mathsf{UFF}(\alpha_n, \mathsf{alive}, T), \mathsf{FF}(\alpha_n, \mathsf{alive}, T)),$ $\alpha_\xi : \Xi(\mathsf{Fract}), \alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$ |
| $p(r)$ | $\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$ |
| $p$ : symmetric$(T)$ | $\alpha_n : \mathsf{Node}_T, \alpha_\omega : \Omega(\mathsf{FF}(\alpha_n, \mathsf{alive}, T), \mathsf{FF}(\alpha_n, \mathsf{alive}, T)),$ $\alpha_\xi : \Xi(\mathsf{LessThan1}), \alpha_A : \mathsf{Asmp}(\alpha_n, \alpha_\xi, T)$ |
| $p(r)$ | $\exists\alpha_g{:}\alpha_\omega.\exists\alpha_k{:}\alpha_\xi.\mathsf{access}(r, \alpha_n, \alpha_g, \alpha_k, \alpha_A)$ |

Figure 5.16: The translation of permission variables, which are syntactic sugar, into the formal language, at both their declaration and use site.

permission variable is rewritten as a series of quantification variables with a fraction classifier $\alpha_\xi$, that is bounded above by LessThan1, and a fraction function classifier $\alpha_\omega$, that is bounded below by FF.

Given such a large difference in syntactic complexity, readers may reasonably wonder whether or not our formal system could have been written to include these simplified polymorphic permissions from the start. Our motivation for presenting polymorphic access permissions in this manner is two-fold. First, we feel strongly that presenting the simplified polymorphic permissions in terms of a formal system where each element of the access permission can be quantified helps in understanding the semantics of the simplified permission bounds. This is particularly true for appreciating the difference between the exact permission and the similar and symmetric permissions. It is crucial to understand that there is some extra level of quantification that is occurring in the latter case that is not occurring in the former case. Second, the full system does allow some specifications that cannot be written in syntactic sugar. For example, if desired, a programmer could force multiple permissions to share the same guaranteed state. Still, due to the large gain in simplicity, we have chosen to implement the simplified syntax directly in our static analysis, described in the next section.

## 5.5 Implementation

In order to better evaluate polymorphic access permissions, we added them to Sync-or-Swim, our protocol checker for the Java language. The entire Sync-or-Swim implementation, which is freely available online[2], is accompanied by Java versions of all of the examples presented in this chapter, which are correctly verified. Sync-or-Swim implements the simplified system from the previous section directly, and does not allow programmers to abstract over each permission element. All of the specifications are written using Java 1.5 annotations.

The following listing is a specification of the `Node` class from our earlier linked list example,

[2] `http://code.google.com/p/pluralism/`

and serves to illustrate the basic form of the Java annotations that can legally be used in our implementation:

```
@Symmetric(value="p",type="T")
@Invariants(@State(name="alive",inv="unique(next) * p(item)"))
class Node<T> {
  @Apply("p") Node next; T item;

  @Unique
  @ResultPolyVar("p")
  T get(int i, int cur) {...}
}
```

The `@Symmetric` annotation introduces a polymorphic permission variable for the scope of the class, which must be associated with a type. The `@Exact` and `@Similar` annotations exist as well, and the permissions introduced have the same semantics presented in Section 5.4. The `@Invariants` and `@State` annotations are already a part of Sync-or-Swim, but now polymorphic permissions variables can be mentioned in these invariants. The `@ResultPolyVar` annotation, along with the `@PolyVar` annotation, allows us to mention permission variables in method specifications. Here is how we might instantiate a similarly specified `LinkedList` class:

```
@ResultShare("Open") Socket
getItemFromList(@Unique @Apply("share(Open)") LinkedList<Socket> l) {
  return l.get(0);
}
```

The `@Apply` annotation applies the share permission kind with a state guarantee of Open to the polymorphic permission parameter of `LinkedList`. At each application site, the applied permission is checked to ensure that it matches the bound on the parameter. Here, since the permission is share, it does. This permission kind and guarantee is subsequently substituted for $p$ in the specification of the `get` method, and the result is that the post-condition of `getItemFromList` is satisfied. In this case, that means that `getItemFromList` returns a share permission with a guarantee of Open.

Our implementation does not support the introduction of permission variables at method scope. All polymorphic permissions must be instantiated at construction time. Unfortunately, Java 1.5 annotations cannot be used on constructor expressions. Therefore a very simple unification algorithm tracks the permissions that are applied to any expression.

Most of the checking functionality piggy-backs on top of the existing Sync-or-Swim tool. Within the scope of a polymorphic variable, a simple flow-based analysis tracks polymorphic permissions as they flow from specification to specification. This analysis treats polymorphic permission variables as being indivisible unless declared as symmetric. As previously mentioned, the analysis also tracks the instantiation of each reference. At method pre- and post-conditions, and receiver pack and unpack sites, this instantiation information is used to determine which permissions are consumed and which permissions are produced. In the case where a polymorphic permission is instantiated with an actual permission, our analysis substitutes the actual permission for the variable in the method specification, and then the original implementation tracks whether or not the appropriate permissions are available in order to satisfy the method pre-condition, and also tracks the newly produced permissions.

## 5.6 Related Work

Existing approaches have contained some similar ideas to the ones presented here, particularly with respect to quantification. In the end, the novelty of our work comes from the manner in which these ideas have been combined, and the novel quantification bounds that we have used to extend modular typestate checking to generic classes.

The original type system upon which this work was based [15] contains a very limited form of quantification. This system allows existential and universal quantification over fractions and fraction functions, but only within the scope of predicates, the syntactic form $P$. This quantification was limited in many ways. Notably, the scope of the quantifiers could not extend over an entire method specification, only within a pre- or post-condition. Our work significantly improves upon the usefulness of the original approach by extending the scope of polymorphism to the method and class level, by allowing state guarantees and assumptions to be abstracted over, and by allowing quantification classifiers themselves to be abstracted over. This last point is what truly enabled the specification and verification of collections that we have seen in practice.

Yasuoka and Terauchi [98] discuss "Polymorphic Fractional Capabilities." Like our work, this approach builds upon Boyland's fractional permissions. Moreover, it supports polymorphism over fractions. Their approach allows a form of polymorphism that supports constraints on the eventually-instantiating fractions. So, for example, a programmer can define a function that takes some fraction of unknown value that must be greater than zero, and it will return that fraction upon completion. This is an improvement over Bierhoff's work, and is a feature supported by our system. Otherwise, the system of fractional polymorphism enabled by this system is quite similar to what it provided by Bierhoff's system, and upon which we improve. Since their language supports no complex data structures, permission cannot be "embedded" inside of other permissions, as we can embed polymorphic permission inside of collection data structures. This is one of the motivations for our approach. Additionally, many of the interesting polymorphic bounds provided in our system are specific to either protocol checking or the five permission kinds supported by our analysis. With only reading and writing permission, some of our new features are not necessary in their system.

Boyland's fractional permissions [25], the basis for Bierhoff and Aldrich [15]'s work, do allow polymorphism, by allowing universal quantification over fractions in procedure specifications. This allows programmers to write procedures that return the same fractions they were given, as long as the procedure body does not depend on them. The main difference is that our work supports a larger number of permission kinds (Boyland's work essentially supports unique and immutable) which means that we must support more interesting sorts of quantification. Boyland's work does not have an analogous notion of polymorphism over fraction and fraction function classifiers, but perhaps could benefit from it. Some of the same concepts apply, for instance their immutable permission is symmetric in our sense of the word.

Higher-Order Separation Logic [17] is able to verify some similar sorts of behavioral properties as our work. For example, using standard logical quantifiers, a function can be defined that is polymorphic in the state of the objects that it accepts and returns. However, existing work does not allow polymorphism over the permission to heap locations. This is not surprising considering that most formulations of Separation Logic have only one "permission." That being said, recent work has extended fractional permissions to separation logic [22]. This work does not permit

quantification over fractions themselves.

In the world of ownership types systems, there has also been some work on polymorphism or "genericity" over ownership relationships. The Generic Universe Type System [36] is one such approach. In this type system, each type parameter of a class is also implicitly associated with a ownership parameter, which makes sense in the context of the system since all types consist of both a class part and an ownership part. Parameters can then be instantiated with a class and ownership type. The rules for "viewpoint adaptation," which are responsible for soundly converting the ownership type of a field to an ownership type relative to its receiver's ownership type, were modified to avoid any unsoundnesses due to covariance in type parameters. This system is based in part upon several earlier generic ownership systems [2, 23, 80]. In fact, their system is quite similar to our own in terms of its functionality. The main difference is really that our language is polymorphic over access permissions, and the associated features of parametricity reflect this. Despite merely being polymorphic over different kinds of aliasing permissions, our system has some small improvements in expressiveness. For example, a generic permission can be soundly used or divided in our system given the right bounds. Or to put it another way, their approach supports only the exact modifier.

Finally, Girard's original work on Linear Logic [48] allowed for quantification over linear facts. However, this work was not presented in the context of managing program resources and therefore it is not clear how this quantification would translate to permission accounting for polymorphic programs.

## 5.7 Conclusion

In this chapter we extended our existing type system, designed to prevent the misuse of object protocols, to allow for polymorphism over access permissions, the static predicates that track what state each object is in, and how those objects may be aliased. This results in increased precision in the specification of classes whose implementations do not constrain the elements they contain, such as a stack that is equally capable of holding unique, open files and shared, open sockets. Our experience, further described in Chapter 6, has shown that this expressiveness is necessary in order to be able to specify commonly used classes without false positives. While this system was expressed in terms of a low-level calculus where each part of an access permission can be abstracted individually, we showed a simplified syntax of our system that can be rewritten in terms of the underlying calculus and described our extensions to the Sync-or-Swim checker in order to support polymorphism.

# Chapter 6

# Evaluation

*Be calm and collected. Peace is a virtue.*

---

This chapter presents a series of case studies designed to evaluate our protocol-checking approach. In each case study, the implementation of our approach, Sync-or-Swim, was used to specify and verify an open-source program. The case studies have various goals, but in general we would like to show that it is *possible* to specify and verify all or most of the protocols that we encounter. We would like to show that this can be done without major changes to the original program, and with a low specification burden. As part of our general desire to not change the original program significantly, we have the specific desire to not add unnecessary synchronization to each program.

The evaluation consisted of two primary case studies and a number of smaller ones. Of the two primary case studies, JabRef was the largest and covered the most number of protocols. JSpider was a second case study designed to help fill a gap in the thread-sharing patterns covered by the JabRef case study. The remaining case studies were done in an earlier phase of this research project, before the final design of Sync-or-Swim was completed. They each show various aspects of the viability of the approach, and in some cases were used to motivate additional features.

In general, the results were positive. In most cases specification and verification was possible with an acceptable rate of false-positives. False-positives were generally confined to testing code, or were simply due to invariants in the programs that are not expressible via the abstraction of a finite state machine. We did find, however, that share permissions could become awkward, forcing us to add unnecessary synchronization and increasing the specification burden. At the end of this chapter there is a discussion of the lessons learned during this process, and how they might be addressed in subsequent work.

In this chapter, Sections 6.1 and 6.2 discuss the JabRef and JSpider case studies, respectively. Section 6.3 discusses some of the smaller case studies we performed, and Section 6.4 contains a discussion of the lessons we learned from the studies, including how we might like to modify our approach in order to address its current deficiencies.

| Program | LOC | Classes | Methods | Description |
|---|---|---|---|---|
| JabRef | 74,217 | 813 | 4,072 | A BibTeX citation organizer |
| JSpider | 8,955 | 187 | 951 | An open-source web robot |

Table 6.1: Basic statistics for the case study programs

## 6.1 JabRef

JabRef is an open-source program that is used to organize BibTeX citations, and is our largest case study. JabRef[1] is a GUI program written in Java. Authors can use JabRef to store a number of citations which can then be easily searched, edited and converted to other formats with a convenient graphical interface. JabRef also contains a number of powerful networked features such as the ability to download citations from popular online databases (e.g., The ACM Portal). At 74,000 lines of Java source (see Figure 6.1) it would probably be considered a medium-sized program. The program has been under development since 2003 and is currently on version 2.6.

JabRef was chosen because of its size and its frequent use of protocol APIs. First, though, a note on how all of the case study programs were chosen is in order. It was our goal to choose the case study programs in an unbiased manner. We did not merely want to select case studies which on which Sync-or-Swim performed well. Unfortunately, some open-source programs may not use APIs that define protocols, they may not use them frequently enough, they may not be multi-threaded, or the APIs themselves may not be used with interesting thread-sharing patterns. All of these possibilities mean that picking programs at random or based on reputation alone may not actually yield an interesting case study.

So we decided to be a bit more selective by using a process that we believe would net us interesting programs without being too biased towards programs that our tool could verify. The first step of this process was to use a code search engine (e.g., Google Code Search) to search open-source code for references to particular APIs. Those APIs were the Timer API and the Socket API, discussed in Section 6.1.1. The use of these two APIs generally implies that the program itself is concurrent (and in the case of the Timer API, is sufficient to make the program concurrent), and both APIs define interesting protocols. So this first step ends up being a great way to filter out programs that are either not concurrent or do not use protocols in any meaningful way. We then manually examined the results for Java programs that had desired characteristics such as program size, and showed extensive use of protocol-defining APIs. For the latter goal, we were able to use the list of protocol-defining types gathered in our empirical study (see Chapter 2). Finally we wanted to be sure that the quality of the code exceeded some basic threshold, so we eliminated any programs that were obviously student projects and examined the code manually to make sure it looked reasonable.

From this process emerged JabRef, which we believe is a great candidate for our evaluation. JabRef is highly multi-threaded and uses a large number of the protocol-defining APIs that we discovered in our empirical study. In fact, of the 813 classes in JabRef, 123 contain calls to protocol-defining methods. The program uses both the Timer and Socket APIs, which

[1]More details on the JabRef program and the source code can be found at the following url: `http://jabref.sourceforge.net/`

as previously mentioned are often intimately related to the thread-sharing patterns of a program. Moreover, approximately 89 classes in JabRef are sub-types of the `Thread` and `Runnable` types, which in Java are the two ways of constructing new threads. So given this, and given that JabRef was approximately the size program we were looking for, it was chosen. We later discovered that JabRef did not contain all the thread-sharing patterns of interest, and this motivated us to add the JSpider case study, discussed later.

### 6.1.1 APIs Verified

For JabRef, Sync-or-Swim was used to verify correct protocol usage of a number of protocol-defining APIs from the Java Standard Library. In each case we started by specifying the methods of each API that would lead to exceptional or undefined behavior if not used correctly. From this specified API, we added annotations to the JabRef code until Sync-or-Swim reported no more errors (or, in the case of false-positives, as few errors as possible). All-in-all we verified seven groups of APIs: Timer, Sockets, Iterator, Streams, Readers, DefaultMutableTreeNode and Others. We will now briefly go through each of these APIs.

**Timer** The timer API consists of the classes `java.util.Timer` and `java.util.TimerTask`. Timers provide a means to execute code (timer tasks) at some point in the future or on a periodic basis. The tasks will always be executed by the timer thread, created by the Java run-time. The timer protocol itself is rather simple: When scheduling a task for execution, an IllegalStateException will be thrown, "if [the] task was already scheduled or canceled." Therefore, we defined four abstract states for the timer tasks (virgin, scheduled, canceled and finished) and specified the schedule methods of the timer so that a task can only be scheduled if it is in the virgin or finished states.

**Sockets** We verified usage of both the `java.net.Socket` and `java.net.ServerSocket` classes. Sockets are used to communicate across a network at a low level of abstraction. Both sockets and server sockets have a notion of being "closed," during which time certain methods cannot be called. But as it turns out, the Socket class defines quite an interesting protocol, with a number of independent states. This can be seen by examining the finite state machine in Figure 6.1 which models its protocol.

The protocol defined by the socket class is much more complicated than most object protocols, which tend to have only a couple of states. (In fact, the model of the protocol in Figure 6.1 is a simplification of the actual protocol, which has many more state transitions.) Sockets are interesting because within the open state there are numerous refinements. The socket can be bound locally to a port, or not, and, independently, it can be connected or disconnected from the remote host. Once connected to a remote host, its reading and writing streams can be shut down independently. All of these restrictions are well-documented in the class' Javadoc description, but with Sync-or-Swim annotations they can be described more formally and more succinctly. A partial specification of this class is given in Figure 6.2.

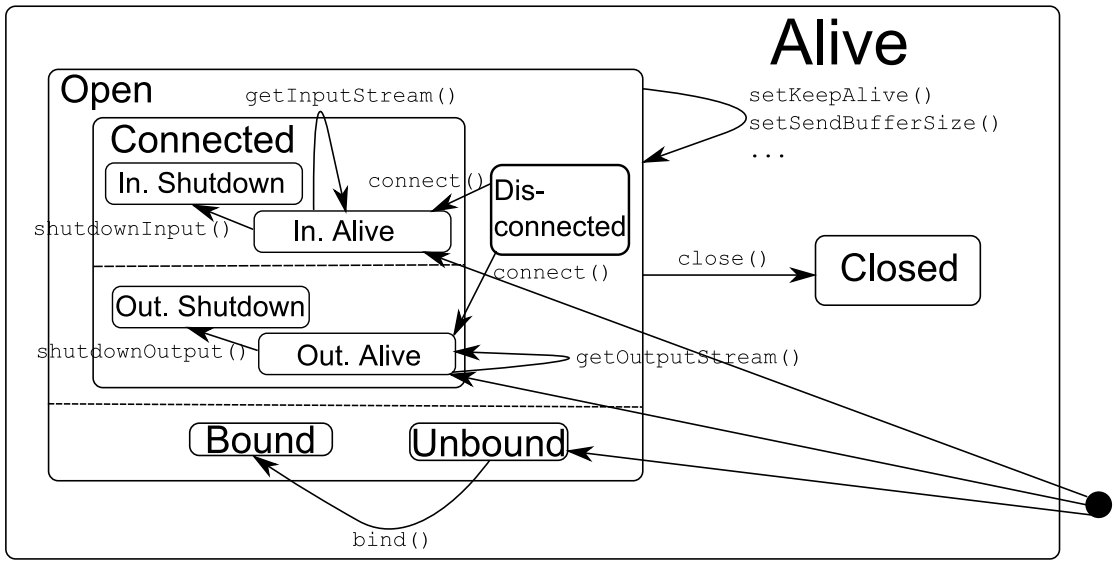


Figure 6.1: A (simplified) model of the protocol defined by the `Socket` class

**Iterator** The iterator protocol has been described extensively in related work [13, 21]. In this protocol, the iterator's `next` method cannot be called unless there are further elements in the collection. We chose to verify use of this protocol because it is widely used and gives us good program coverage, but it is not extremely interesting.

**Streams and Readers** The stream classes, `java.io.InputStream` and `OutputStream` and their subclasses, each define a straightforward open/closed protocol. Streams can be closed, at which point most methods defined by the class can no longer be called. Streams are typically used for reading and writing text and data. They are made somewhat more interesting by the fact that several subclasses (e.g., FileInputStream, BufferedInputStream) are commonly used, which will allow us to evaluate the subtyping features of our specification language. Steams are also interesting because of their interaction with the Socket classes previously mentioned.

The reader class, `java.io.Readable` and its subclasses, are quite similar to the stream classes. These classes are in fact wrappers around data sources such as streams, and provide a more convenient means of accessing the data. Like streams, they define an open/closed protocol.

**Other Protocols** JabRef contained uses of a number of other protocol-defining classes from the Java Standard Library. The `Enumeration` interface is quite similar to the iterator interface, and defines a nearly-identical protocol. It was used in several places throughout JabRef. A zip file, as reified in the class `java.zip.ZipFile`, provides a means of reading and writing compressed files. It defines a simple open/closed protocol. Notably, normal files (i.e., the java.io.File class) do not define a protocol. The same goes for the class `java.net.URLConnection`. This class can be used to connect to and read from web resources indexed by a URL. It defines a connected/disconnected protocol, where many, but not all methods, are illegal to call when the instance is not connected to a remote location. The `Element` class in the package `org.w3c.dom`

```
@Refine({
  @States({"Open","Closed"}),
  @States(dim="local", refined="Open", value={"Bound","Unbound"}),
  @States(dim="remote", refined="Open",
          value={"Connected","Disconnected"}),
  @States(dim="input", refined="Connected",
          value={"InputOpen","InputShutdown"}),
  @States(dim="output", refined="Connected",
          value={"OutputOpen","OutputShutdown"})
})
public class Socket {
  @Perm(ensures="unique(this!fr) in InputOpen,OutputOpen,Unbound")
  public Socket(InetAddress address, int port);

  @Full(requires="Disconnected",ensures="InputOpen,OutputOpen")
  public void connect(SocketAddress endpoint);

  @Full(requires="Unbound",ensures="Bound")
  public void bind(SocketAddress bindpoint);

  @Full(requires="Open",ensures="Closed")
  public synchronized void close();

  @Full("InputOpen")
  public InputStream getInputStream();

  @Full("OutputOpen")
  public OutputStream getOutputStream();

  @Full(requires="InputOpen",ensures="InputShutdown")
  public void shutdownInput();

  @Full(requires="OutputOpen",ensures="OutputShutdown")
  public void shutdownOutput();

  @Full("Open")
  public void setKeepAlive(boolean on);
  ...
}
```

Figure 6.2: A (simplified) specification of the `Socket` class

is the basic type of the XML type hierarchy. XML nodes, most of which descend from `Element` can be either mutable or immutable, and certain features of a node (e.g., XML attributes) cannot be modified when the element is immutable. The mutability property is set at construction time, so we consider it to be a "type qualifier" protocol (see Chapter 2).

Finally, the class `DefaultMutableTreeNode`, from the package `java.swing` is used in a few places in JabRef. It defines a general-purpose tree data structure and is used by many of the classes in the Swing GUI framework. This class defines a simple protocol wherein some in-

stances are allowed to have child nodes and others are not. At run time, the `setAllowsChildren` method can be called to change whether or not a particular instance is allowed to have children. Attempting at add children to a tree node that does not allow children will result in a run-time exception being thrown. This protocol ended up being quite interesting because its protocol was very *uninteresting* with respect to JabRef and yet verification of its use was quite tricky. This issue will be discussed in detail in the next section.

### 6.1.2 Program Architecture and Thread-Sharing Patterns

To understand the extent to which Sync-or-Swim was successful, it is helpful to understand the threading architecture of the JabRef program. JabRef shares an overall program architecture with many other GUI programs. At the outset, the program sets up a number of visual elements like buttons and windows and creates call-back routines that will be executed when the user takes specific actions. Then, as part of the start-up phase, the main BibTeX file, which acts as a permanent representation of the program's state, is parsed and the data loaded into in-memory database objects. At the end of the start-up phase, the program is displaying bibliographic entries and is ready to respond to user input. JabRef is built on the Swing GUI framework, and from this point forward, the Swing event thread is where much of the action occurs. It handles keyboard and mouse events and invokes the appropriate call-back functions defined at startup time. These functions either act directly on the database entries and change the graphical layout within the event thread, or spawn new worker threads to carry out long-running and asynchronous tasks.

This high-level overview is an important, if simplistic, view of the threading architecture of of JabRef, but it is also helpful to describe some specific examples of thread-sharing, particularly as they apply to the APIs that we are attempting to verify. In practice, two main patterns of thread-sharing were used throughout JabRef. The first pattern was of an object being used as a thread-local object (i.e., completely unshared). The second pattern was of a thread-shared object being modified by multiple threads but never leaving one particular abstract state. These were really the only two patterns that we needed to consider for the purposes of the APIs we wanted to verify. The former pattern could generally be specified using a unique permission. The latter we specified using a share permission with a state guarantee. We believe that these patterns are likely the most common for protocol-defining objects. This is what we have observed in practice, and it also makes sense, because trying to obey a protocol on a completely thread-shared object is a difficult proposition. The code will either have to insert numerous state checks or will have to enforce the protocol through more subtle invariants. That being said, we do believe that there is at least one more important and likely thread-sharing pattern, that of ownership transfer. In this case, a protocol-defining object is created by one thread and passed to another thread. The desire to evaluate Sync-or-Swim against this pattern motivated our selection of the JSpider case study.

The `JSTORFetcher` class is a representative example from JabRef of a protocol-defining object being used in a thread-local manner. This particular class is responsible for fetching BibTeX entries from the JSTOR digital archive.[2]. This class makes use of the `URLConnection` class, described earlier, to retrieve BibTeX entries from a web address. The instance of this class

[2] `http://www.jstor.org`

that is created by the JSTOR fetcher is used in an thread-local manner, as seen in Figure 6.3.

The method `getBibtexEntries` is an instance method of the `JSTORFetcher` class. This method opens up a remote connection, downloads the BibTeX entry and parses it. It is called (indirectly, through `processQuery`) by the thread created in the `actionPerformed` method, also shown in Figure 6.3. This method is called, and hence the thread created, in response to a user's action. The important thing to note is that even though the connection is created through a newly created thread, the `URLConnection` instance is treated thread-locally. A unique reference is created (line 4) which is used to connect (line 6) and read (line 9). Because the object is not thread shared, we can be sure that the object is still connected when the call to `getInputStream` is made, as required.

```
1  Collection<BibtexEntry> getBibtexEntries(String ticket, String citations)
2  {
3    URL url = new URL(URL_BIBTEX);
4    URLConnection conn = url.openConnection();
5    conn.setRequestProperty("Cookie", ticket + "; " + citations);
6    conn.connect();
7
8    BibtexParser parser = new BibtexParser(
9        new BufferedReader(new InputStreamReader(conn.getInputStream())));
10   return parser.parse().getDatabase().getEntries();
11 }
12
13 void actionPerformed(ActionEvent e) {
14   ...  // sets up import dialog
15
16   new Thread(new Runnable(){
17     public void run(){
18
19       if (fetcher.processQuery(tf.getText().trim(), dialog, frame)){
20         dialog.entryListComplete();
21       } else {
22         dialog.dispose();
23       }
24     }
25   }).start();
26 }
```

Figure 6.3: The `getBibtexEntries` method of the `JSTORFetcher` class along with the event call-back that spawns the fetching thread.

There are a great deal more uses of thread-local objects throughout JabRef, as one might imagine. In particular, most of the uses of the iterator and enumeration protocols are used in a thread-local manner.

For examples of the latter sharing pattern, in which mutable objects are thread-shared but never leave on particular state, we can look at the `StreamPrinter` class. This is a simple active class that holds references to two streams, an input stream and an output stream. When it is run,

```
@Invariants(@State(name="alive",inv="share(stream,Open) * share(out,Open)"))
public class StreamPrinter implements Runnable, Stoppable {
  // The input stream to read from.
  private InputStream stream;
  // The print stream to redirect to.
  private PrintStream out;

  @Perm(ensures="unique(this!fr)")
  public StreamPrinter( @Share(guarantee="Open",returned=false)
    InputStream s, @Share(guarantee="Open",returned=false) PrintStream p ) {
    stream = s;
    out = p;
    thread = new Thread( this );
  }

  public void run() {
    int buf;
    boolean me;
    while ( !stopped && ( buf = stream.read() ) != -1 ) {
      synchronized( this ) {
        me = flush;
      }
      synchronized( out ) {
        out.print( (char)buf );
        if ( me ) out.flush();
      }
    }
  }
}
```

Figure 6.4: The (simplified) `StreamPrinter` class whose state invariant guarantees that both streams remain open

by calling the `start` method, a newly spawned thread will read from the input stream and write to the output stream. While the streams themselves define a simple open/closed protocol, the reality is that the streams given to the stream printer are never closed during the lifetime of the application. They are closed by the Java runtime upon finalization. Some of the streams that are passed to the stream printer are thread-shared (standard in and standard out, for instance, are accessed by other threads). As mentioned, the solution is to guarantee the streams in the "Open" state as part of the state invariant of the `StreamPrinter` class. The result can be seen in Figure 6.4.

This pattern was observed multiple times during the JabRef case study.

### 6.1.3 Results

The results from our experiment show that Sync-or-Swim was generally satisfactory for the specification and verification of the APIs in question and did not impose unreasonable burden in most cases. In this section we will describe the results, starting with a discussion of Sync-or-Swim's

| API | Annotations | | | Spec. | | False | Sync. |
|---|---|---|---|---|---|---|---|
| | **Invariant** | **Method** | **Poly.** | **Time** | **Warnings** | **Positives** | **Added** |
| Timer | 2 | 2 | 0 | 1h | 4 | 4 | 0 |
| Sockets | 2 | 6 | 0 | 30m | 2 | 0 | 2 |
| Iterator | 8 | 16 | 0 | 8h | 28 | 26 | 0 |
| Streams | 6 | 23 | 0 | 4h | 0 | 0 | 5 |
| Readers | 4 | 36 | 0 | 105m | 0 | 0 | 0 |
| Tree | 64 | 178 | 1 | 9h | 26 | 26 | 56 |
| Others | 0 | 8 | 0 | 5h | 1 | 1 | 0 |
| **Total** | 86 | 268 | 1 | 29h | 61 | 57 | 63 |

Table 6.2: Results from the JabRef case study

ability to express the protocols in question, and then proceeding to remarks on the annotation burden and explanation of the true and false positives. Table 6.2 succinctly summarizes these results.

### Specification

The ability of Sync-or-Swim to specify the protocols and sharing patterns we encountered generally was quite sufficient. The specification process can be considered in three parts. First, there is ability of the tool to simply express the protocols that clients of an API must obey. Next, we consider the ability of the tool to specify the state invariants, which are necessary to verify the implementation of protocols or to verify use of types that are references via fields. The last part is the ability of the tool to express the thread-sharing and aliasing patterns that occurred in the case study programs. This section will generally focus on examples where the specification language was successful. Since many of the imprecisions we encountered could be considered to be failures in specification, places where the specification language was not so successful will be discussed along with the false positives.

Kevin Bierhoff's Ph.D. thesis [13, pp. 85-93] largely established that the specification language of Plural was expressive enough to specify just about every protocol from the client's perspective. Sync-or-Swim uses the same specification language and our experience confirmed his findings. We never had trouble expressing the protocols defined by the APIs under discussion. Even when those protocols were intricate or contained a large number of states our experiences with specification were good. In particular, we credit the state refinement and state dimension features of the model with much of our success. These turned out to be enormously powerful metaphors that enabled the expression of just about any protocol. (For examples of sub-states and state dimensions, one need only to turn to the `Socket` specification previously described in Figure 6.2.)

As far as state invariants go, Sync-or-Swim generally did a fine job. For the most part, in JabRef there was not much need to write particularly interesting state invariants. Many times state invariant specifications were only used to signify that a class always had some permission to a field. Often the state of the field was irrelevant, or when it was, it was often guaranteed to be in some particular state, such as the open stream fields in the `StreamPrinter` class (described

earlier in Figure 6.4). Another common pattern we named, "protocol propagation." In this pattern a class defines a protocol that is similar or identical to the protocol defined by one of its fields. This often happens when a class is acting as a wrapper or delegate for a protocol-defining type, interpreting the method calls and passing them on to the wrapped field as appropriate. The `IteratorV` class, which wraps an iterator for pair objects and only returns the second value, is a good example of this pattern and can be seen in Figure 6.5.

```
@Invariants({
  @State(name="alive", inv="unique(iterator) in alive"),
  @State(name="HasNext", inv="iterator in HasNext"),
  @State(name="CanRemove", inv="iterator in CanRemove")
})
class IteratorV<V2> implements Iterator<V2> {
  private final Iterator<? extends Pair<?, V2>> iterator;
  ...
}
```

Figure 6.5: State invariant specification for the `IteratorV` class, which propagates the protocol from the `iterator` field

One minor recurring problem is that we cannot specify anonymous classes with the `@Invariants` annotation. This is partially due to the design of Java annotations which do not allow annotations on anonymous classes. But even if the syntax of Java would allow it, we would have to find some way to specify the permission captured to references from an outer scope. This was left for future work, and as a result, during the case study we were forced to manually rewrite anonymous classes to local classes, and turn any captured references into references explicitly passed to the object constructor.

At a high level, the five permission kinds did a pretty good job approximating actual thread-sharing permissions. In practice, if one just considered unique to be thread-local and share to be thread-share, the specifications work quite well. This is not to say that we were never required to add unnecessary thread synchronization. In fact we were, many times. But this was more due to typing rules that were unnecessarily strong, rather than a failure of specification.

One potential concern with our specification language is the programmer's inability to specify any lock other than `this` as being responsible for protecting the internal state of an object against concurrent modification. However, this never was an issue in our case study. The ability to specify another lock as protecting an object's state was never required.

### Burden

Table 6.2 summarizes the number and type of specifications that were required to be written in JabRef in order to check each API. These annotations are broken down into three categories. "Invariant" annotations are used to specify state invariants. "Method" annotations are used to specify pre- and post-conditions, and "Poly." annotations are used by our polymorphic extension. They are used to instantiate a polymorphic API with a specific permission.

Table 6.2 also summarizes the amount of time taken to specify and verify the code under the column heading, "Spec. Time." In this column, 'h' signifies hours and 'm,' minutes. Note that

these numbers are *highly* approximate as they were estimated from source code repository logs and other indirect evidence. Still, they give a rough idea of the overall burden on the programmer. Most APIs took either about an hour or about half of a working day. The two exceptions were the `Iterator` API and the `DefaultMutableTreeNode` API, each of which took an entire working day.

The iterator API was not a particular burdensome API to check. With a few exceptions it was used in a thread-local, method-local manner. However, the fact that it was extremely widely used throughout JabRef means that there were just more use sites that potentially needed to be annotated with additional permissions. Furthermore, we encountered a few bugs in the Sync-or-Swim implementation during this part of the case study, and while the 8 hour figure does not include time taken to fix the bugs, it does include some time spent diagnosing the issue, since we had to be sure the problem was not with our specification.

The mutable tree API is an entirely different story, however. It was more painful to specify and verify. This difficulty is made all the more disappointing by the fact that the protocol was a simple one and used in an entirely uninteresting manner in the JabRef program. As a reminder, the `DefaultMutableTreeNode` class is a simple reusable data structure for building trees. Each node in the tree either allows children or does not. Whether a node allows children or not can be changed at run-time. For tree nodes that do not allow children, certain method calls are forbidden, such as the call to `add`. In JabRef, every instance of this class allows children. The specification strategy was to use a share permission, guaranteed in the, "AllowsChildren" state throughout. Such a permission, if it is the only permission to be used, it quite flexible, since it can modify and can be duplicated innumerable times without having to worry about the possibility of modification from other threads. At first blush, the task of specifying the use of the API within JabRef seemed quite easy.

However, this was not the case, and as we will demonstrate, caused us to realize that when types with specified methods are used as fields, the amount of specification work that must be done increases by several times. Here is what happened: First, the API itself is not widely used in JabRef. Only seven classes reference the type `DefaultMutableTreeNode`. However, those classes are part of a larger cluster of classes, meaning that they are referenced both directly and indirectly, by a large number of classes. And importantly, they are quite often referenced as fields. When a type, some of whose methods require permission, is referenced as a field it begins a specification process like the following:

1. See what permission is required of the field in order to make the required method calls, and determine if the outer class defines any of its own states.
2. Create or modify a state invariant for the outer class so that the proper permission is held to the field.
3. Specify the constructor of the outer class as returning a permission to the newly constructed object.
4. Specify the method of the outer class which contains the field access as requiring a permission to the receiver.
5. Find all callers of this method and see if those callers now require annotation, since a new permission burden has just been created.

If the callers of the newly specified method are calling that method on a field, as was frequently the case for the transitive clients of the mutable tree API, the process repeats over again requiring more and more specification of methods and invariants. A method may have numerous callers, necessitating a specification process that proceeds in a tree-like manner. The result is a frustrating process which seems to never end, since each time an annotation is added, it may necessitate more annotations. This can also mean that the number of specifications that are required to verify one particular API can dwarf the actual calls to that API. JabRef contains 33 calls to `add` and `insert`, the two methods of the API for which some state must be established as a pre-condition. But verifying these calls required 243 annotations, one quarter of which were state invariant annotations. In our case much of the frustration could have been alleviated with good default permissions, a topic we will discuss further in Section 6.4.

All in all, 357 annotations were required to verify seven APIs used in a program of 74,217 lines, giving a specification density of 1 annotation per 207 lines of code.

### Required Modifications

Our goal in this case study was, to the extent possible, to not modify the code in the programs under study. Our hope was that Sync-or-Swim would be able to verify large programs as they exist "in the wild," even if that means supporting difficult patterns or features of Java. But in some places, we could not achieve this goal, and JabRef could not be verified without changing the program. Here we discuss some of these, mostly minor, changes.

One of our initial concerns when designing our approach was that access permissions, designed to control aliasing, would not be a precise enough abstraction to specify the thread-sharing patterns in question, and that the result would be lots of unnecessary thread synchronization. The "Sync. Added" column in Table 6.2 summarizes the number of times that we were required to add synchronization (specifically, `synchronized` blocks) to JabRef. While a few races were found, the majority of these changes were unnecessary. They could be considered false positives of our analysis. They mostly occurred while trying to verify the mutable tree API, which due to its extensive use of guaranteed, share permissions, required the synchronized modifier on just about every method that accessed a permission-holding field. For the other APIs, though, Sync-or-Swim performed quite well, and in over half of the APIs verified, required no additional synchronization.

There were a few other modifications to the code that were done on a regular basis and mostly in response to bugs in the Sync-or-Swim implementation, which we now summarize:

**Anonymous Classes Converted (5)** As previously mentioned, Java annotations cannot be used on anonymous classes. So, when a anonymous class needs a state invariant, it must first be converted to a local class. This modification does not change the program behavior.

**Local Variables Extracted (15)** Due to a bug in the local alias analysis of Sync-or-Swim, it was occasionally necessary to use the "extract local variable" refactoring in order to verify a particular piece of code. More concretely, in 15 locations there were method call sites containing non-trivial subexpressions, and we replaced them with variables initialized to the value of those

subexpressions. The exact cause of these bugs is an architectural limitation of our implementation, and is not a fundamental limitation. Such bugs would not occur in an implementation where the local, must-alias analysis had access to permission information. Briefly, the bug causes Sync-or-Swim to lose track of object locations when subsequent method arguments contain method calls. While such a refactoring can in general change a program's behavior, in all of the cases where it was employed it did not.

### True and False Positives

Overall Sync-or-Swim reported 61 warnings on the JabRef case study, the vast majority of which occurred while verifying the iterator and mutable tree APIs, and the majority of which were not real bugs. In this section we will show a few of the actual bugs we encountered as well as provide explanation for the false positives.

All of the warnings we encountered while attempting to verify the iterator API occurred because of a call to the `next` method without a proceeding call to the `hasNext` method. In two cases these warnings represented actual bugs that could occur at run-time. Figure 6.6 shows the first bug, but the second is quite similar. In this example, a BibTeX entry in the form of a string is passed to a parsing method. This string comes from the user and therefore may not parse. The `fromString` method returns a collection, and since `singleFromString` only expects one entry to be present (or does not care about the others) returns a call to `iterator().next()` on the collection, returning just the first element. Unfortunately, if the BibTeX string fails to parse, the `fromString` method will return an empty collection rather than a null value, and in this case the code will throw an exception. In the other bug, `iterator().next()` is called on a collection that is parsed from a remote URL. Since it is unknown whether the URL will even exist in the future, it is unreasonable to expect it to always return a parsable BibTeX entry.

```
public static BibtexEntry singleFromString(String bibtexString) {
  Collection<BibtexEntry> c = fromString(bibtexString);
  if (c == null){
    return null;
  }
  return c.iterator().next();
}
```

Figure 6.6: A bug in JabRef; The BibTeX string may fail to parse, at which point the collection will be empty, rather than null

Of the remaining 26 warnings, 19 occurred in unit tests, where the failure of a call to the `next` method would have indicated some larger bug in the JabRef application. Four other warnings occurred in code where the programmer had established invariants ensuring that the iterated collection would be non-empty. And in the remaining three cases, a dynamic check was performed on the collection to see if its size was greater than zero, which ensures that a single call to the `next` method is okay. Sync-or-Swim currently does not support reasoning about integer values.

While verifying the stream API, Sync-or-Swim issued two warnings which we believe are indicative of a protocol-based race condition. The situation is illustrated in Figure 6.7 which

contains a (simplified) version of the `RemoteListener` class. This class is a thread that allows separate instances of JabRef running on the same computer to send commands to this, the currently running instance. In the figure we can see two methods, the `run` method which is executed by the remote listener thread and the `disable` method which is called by a GUI thread in response to a user changing the program preferences. This has the effect of closing the open socket that the remote listener uses to communicate. The method also sets the field `toStop` to true.

```
1  public class RemoteListener extends Thread {
2    // ...
3    @Share(use=Use.FIELDS)
4    public void run() { // warning
5      while (active) {
6        try {
7          Socket newSocket = socket.accept(); // warning
8          // ...
9          if (toStop) {
10           active = false;
11           return;
12         }
13         // ...
14       } catch (SocketException ex) {
15         active = false;
16       }
17   }
18
19   @Share(use=Use.FIELDS)
20   public void disable() {
21     toStop = true;
22     socket.close();
23     // ...
24   }
25 }
```

Figure 6.7: The `disable` method is called by a separate thread and can be called at any time, since it is called in response to user events. As a result, the socket's `accept` method may be called when the socket is not actually open.

By examining the body of the `run` method, we can see that there may be some problems. In addition to the data race on the `toStop` field, the remote listener thread only checks the `active` field before calling `socket.accept()` (line 7) which requires that the socket be in the open state. The `toStop` field is only checked after the the `accept` method is called. The result is (and as is indicated by warnings on line 7 and 4) that the GUI thread may call the `close` method on the socket before the `accept` method is subsequently called, resulting in a run-time exception of type `SocketException`. There is some indication that the programmers themselves expect this outcome, since the `active` field is set to false when the `SocketException` is thrown. Still, one wonders if there may have been confusion on the part of the programmer. A fix could be enacted by synchronizing both the `run` and the `disable` methods, and additionally moving the check of the `toStop` field on line 9 above the current line 7.

We also found two data races during the course of our case study. The first data race was on the `stopped` field of the `StreamPrinter` class. This field is accessed by multiple threads concurrently and some of the accesses are writes. The programmer clearly knows that the field is accessed by multiple threads since in the implementation of the `stop` method, which modifies the field, a lock is used. However, in the `run` method of this class (it is a thread), even though two separate locks are acquired, the read of the `stopped` field does not occur within either of them. It should be noted that Sync-or-Swim did not directly find this race. Since the `stopped` field was not used as part of a state invariant, Sync-or-Swim does not attempt to enforce proper synchronization. It was, rather, during the close inspection required of any class under verification that the race was noticed.

The second data race, however, was found by Sync-or-Swim. This race occurred in the `RemoteListener` class on the `toStop` field. In this class, a GUI thread can tell the listener to stop listening by calling the `disable` method, setting the field to "true." The listener thread then notices the field has changed in the `run` method. Unfortunately, no synchronization is used, and the field is not marked as volatile. This was detected by Sync-or-Swim because both threads required share permission to the `RemoteListener` instances. Sync-or-Swim complained during the read and write to the `toStop` field that an object was being unpacked with share permission but that no synchronization was used.

While prior existing static race detection tools would have found the two data races, they would not have found the race on the abstract state of the socket. The remaining warnings were false positives and fit into one of a few categories:

**Sync-or-Swim Loop Bug (12/57)** A number of the false-positives came from a bug in Sync-or-Swim's handling of joining share permissions. Recall that the analysis is implemented using a lattice. In certain situations, when multiple share permissions of different fractional values are merged together, rather than the result being a share permission of unknown fractional value, the result is a permission with unsatisfied constraints. This particularly happens in loops. We consider this to be an implementation bug, and nothing conceptually prevents these cases from being correctly verified.

**Field of Other Receiver (9/57)** Currently Sync-or-Swim only supports unpacking the receiver of the current method (i.e., `this`). In these nine cases, a field was accessed from an object other than the receiver of the current method, and therefore could not be properly verified. This is not a fundamental limitation, but merely a missing feature. We believe that such a feature could be added with minimal effort.

**Unique Permission to Static Field (4/57)** Some of the false positives were due to the need to associate a static field with a unique permission. This is not a feature that is supported by Sync-or-Swim, because we have not yet figured out the best way to specify and verify the feature (verification would require a process analogous to unpacking but for classes rather than instances). Still, we can say confidently that a unique permission is appropriate, since in all four cases the field was assigned from a newly created object, accessed in a manner that required unique permission, and then not subsequently accessed in a manner that required any

permission.

**BasePanel (4/57)** Due to time constraints and previous experience, we did not follow the specification of the `DefaultMutableTreeNode` class as it propagated into the `BasePanel` class. This decision resulted in four false positives. We regretfully came to this decision during our verification of the `DefaultMutableTreeNode` API. We discovered that, due to the `BasePanel`'s numerous references to other classes in JabRef, adding specifications to the methods of this class eventually requires a great deal more specifications to be added to a larger number of other classes in the program. We spent about two days adding new specifications to the program before realizing that we were not reaching a solution any time soon. This issue is mentioned in the discussion section.

**Unexpressed Invariant (1/57)** In one case the `nextElement` method of the `Enumeration` interface was called without a proceeding call to the `hasMoreElements` method because the programmer had previously established that the underlying collection was non-empty.

**Circular Initialization (1/57)** In one case Sync-or-Swim issued a warning due to the circular initialization of the fields of a class. This sort of pattern was encountered a few times in our case studies and cannot currently be verified by our methodology. Other approaches, namely Delayed Types [40] and Masked Types [82], can successfully verify similar patterns. At issue is the `GroupSelector` class, whose invariant for the "alive" state guarantees share permission to a number of its fields. The objects pointed to by the fields, which are created in the constructor of `GroupSelector` require their own reference back to the `GroupSelector` that is being constructed. This creates a sort of "Chicken and Egg" problem where each field needs a permission to the `GroupSelector` in the "alive" state to be constructed but no permission is available because the `GroupSelector` instance must construct those field in order to be in the "alive" state in the first place.

**Iterator (26/57)** (Previously discussed)

**False Positive Rate** Overall, the rate of false-positives for this 74,000 line program is one per 1298 lines of code.

## 6.2 JSpider

JSpider is the second of our two primary open-source case study programs. JSpider[3] is freely available web spider engine written in Java. It can be used by those who are interested in crawling the web to obtain data, for example as the basis for a search engine. At around 9,000 lines of source, it would be considered a relatively small program. It was released in 2003 and work on the project is largely complete.

[3]More details on the JSpider program and the source code can be found at the following url: `http://j-spider.sourceforge.net/`

As mentioned in the previous section, JSpider was chosen as a case study to address the absence of what we see as an important pattern of thread-sharing, that of ownership transfer from one thread to another. This pattern did not occur in JabRef. JSpider was chosen because it exhibits this pattern. For the purposes of this case study we only attempted to verify one protocol, a protocol defined by the application itself.

### 6.2.1 The WorkerTask API

The goal of this case study was to verify correct use of the `WorkerTask` API inside the JSpider application. Worker tasks are jobs that will be executed by a worker thread. The `WorkerTask` interface defines three methods of interest, `prepare`, `execute` and `tearDown`. The `execute` method is the primary method defined by the interface. When this method is called by the worker thread, the important action that the implementing class was designed to carry out should take place. However, the interface defines two additional hooks, `prepare` and `tearDown`. These methods allow implementing classes a chance to have some code executed either before or after the main task itself is executed.

The contract of the interface is that the methods will be called in-order, `prepare`, `execute` and `tearDown`. Therefore, we intend to verify that the framework itself always calls the methods in this order. If a class is implemented in such a manner that it depends on this property, but the framework calls the methods out of order, it may cause the class to function improperly. We want to ensure that this does not happen.

```
public interface WorkerTask extends Task {

  public static final int WORKERTASK_SPIDERTASK = 1;
  public static final int WORKERTASK_THINKERTASK = 2;
  public int getType ( );

  /**
   * Allows some work to be done before the actual Task is carried out.
   * During the invocation of prepare, the WorkerThread's state will be
   * WORKERTHREAD_BLOCKED.
   */
  @Unique(requires="Init",ensures="Prepared")
  public void prepare ( );

  @Unique(requires="Prepared",ensures="Executed")
  public void execute();

  /**
   * Allows us to put common code in the abstract base class.
   */
  @Unique(requires="Executed",ensures="TornDown")
  public void tearDown ( );
}
```

Figure 6.8: The `WorkerTask` interface, complete with annotations

The specification for the `WorkerTask` API is given in Figure 6.8 and it is fairly straightforward. Each method requires a unique permission to the receiver in order to be called. In order to call the `prepare` method, the task must be in the "Init" state, which will be given by the constructor of implementing classes. Afterward the object will be in the "Prepared" state, which is required in order to call the `execute` method. After this, the receiver will be in the "Executed" state, which is required in order to call the `tearDown` method.

This choice of protocol is interesting because it requires us to verify the implementation of the library itself rather than clients of a library. In all of the APIs we verified in the JabRef case study, we really wanted to make sure that clients of were obeying the protocol defined by the API. Here we are making sure that the infrastructure, that is the task queue and worker threads, are living up to the contract defined by the interface. If they do, we will know that clients are guaranteed the expected behavior when they implement the interface.

In fact, these sorts of methods on an interface, which define call-backs into a client's code at various points in an application are known as "life-cycle" methods, and they are quite common in software frameworks that use a plug-in architecture. Therefore this interface provides Sync-or-Swim with an interesting and relevant challenge. Finally, note that Sync-or-Swim is only capable of specifying that the methods are called in the correct order. It cannot guarantee that the methods are ever actually called.

### 6.2.2 Program Architecture and Ownership Transfer

We are primarily interested in JSpider because of the means by which worker task objects are passed from one thread to another. The architecture of the tasks code shows why such patterns occur. Tasks can be one of two types. "Spider" tasks require information to be fetched from a remote web site. "Thinker" tasks perform jobs that do not require the fetching of remote data, for example interpreting fetched data and making various decisions about how to proceed. Both types of tasks are created by implementing the `WorkerTask` interface. `FetchRobotsTXT-TaskImpl` and `InterpreteHTMLTask` are two examples of worker tasks. The former fetches the robots.txt file found on many web sites, and the latter is an extensible HTML reader that can be used to, for example, find all of the links on a page.

As execution proceeds, a number of worker threads are created. These threads are assigned a task by a worker thread pool object which, when given a task, polls all of the threads for an available thread. When it finds one, it gives the task object to the thread for execution. The worker thread, which is running concurrently, periodically checks to see if a new task has been assigned to it. When it finds that one has been assigned, it calls the three lifecycle methods on the task, and returns to a dormant state. While the thread pool is responsible for handing tasks off to worker threads, it is the scheduler that is responsible for storing the tasks in a queue until they are ready to be executed. This basic architecture lays out at a high level why ownership transfer is an appropriate metaphor.

Ownership of objects is transferred from threads assigning tasks to the worker threads that execute the tasks. When each new task is created, it is created local to one thread. Any initialization that needs to take place is done in the creating thread. The task is then handed off to the worker thread via the thread pool, and no permission is retained by the creating thread. This is a case where the unique permission can be applied, even though the object is shared between

| API | Annotations | | | | Spec. | | False | Sync. |
|---|---|---|---|---|---|---|---|---|
| | Invariant | Method | Poly. | Other | Time | Warnings | Positives | Added |
| WorkerTask | 2 | 30 | 2 | 2 | 6h | 6 | 4 | 2 |

Table 6.3: Results from the JSpider case study

multiple threads. For verification purposes, the important thing to establish is that each task is in the "Init" state when it is handed off to the worker thread. By using a unique permission, we are guaranteed locally that the state of the task is not being modified by other threads. And in practice, the unique permission worked very well for this ownership transfer pattern.

### 6.2.3 Results

In our experience, Sync-or-Swim did an excellent job verifying that the framework called the lifecycle methods in the correct manner without being especially burdensome. The sole row in Table 6.3 summarizes our results.

#### Specification and Burden

Table 6.3 lists the number of annotations that were required to be added to the JSpider program in order to check the API. As previously mentioned, the unique permission was the basis for most of the specification work we did. The unique permission in our approach is interpreted as thread-local, but unique permissions can also be passed from thread to thread. This was reflected in the final specification count where `@Unique` and `@ResultUnique` annotations made up 21 of the 30 method annotations we used.

In one case the use of polymorphic permissions was required. The `SchedulerImpl` class implements the scheduler abstraction which is responsible for storing the worker tasks until they are ready to be executed by a worker thread. Since we need to ensure that they are always in the "Init" state right up to the point where they are given to the worker thread, the state invariant for the work queue must record that for each element of the list there is a unique permission in the "Init" state. Instead of hard-coding this (and because the same list class is used in different ways in other parts of the program) we made the list polymorphic. When instantiated inside of the scheduler implementation, the `add` method consumes a unique permission and the `remove` method produces one. Elsewhere, no permission is required when calling these methods.

Two "Other" annotations were also needed: In one case a `@NonReentrant` annotation was needed to verify a class that by design would never make reentrant calls. In another case, a `@ForcePack` annotation was used to tell Sync-or-Swim which state to pack an object to. In this particular case, the packing inference algorithm did not pack the object early enough to avoid an imprecision due to merging.

Sync-or-Swim was able to specify all of the protocols and aliasing and thread-sharing patterns we encountered in JSpider with one exception. It currently does not support the ability to specify permission to the *contents* of an array (as opposed to the array object itself). This lead to a few false positives, as discussed later. Due to the prevalence of unique permissions, we were also

not required to add any additional synchronization. The entire process of selecting, specifying and verifying the JSpider case study went quickly, and took a little over half of a work day.

Finally, we should note that for multi-threaded programs it was quite helpful to support implications in state invariants between Boolean fields and permissions to fields. By this we mean that in certain situations, the truth or falsehood of a Boolean field implies that the receiver object or a different field is in a particular state. As an example, consider the state invariant specification for the `WorkerThread` class, shown in Figure 6.9. Its specification says that whenever the `assigned` field is true, the class will have a unique permission to the task field and that object will be in the "Init" state. This invariant holds because when the `assign` method is called by the thread pool, a unique permission is given and the `assigned` field is set to true. In the thread's run method, if `assigned` is true, then the thread knows it is safe to call the `prepare` method on the task, which requires the task be in the "Init" state. By the end of the method the task is no longer in the "Init" state, but `assigned` is cleared and so the permission need no longer hold. We noticed a similar sort of invariant in several of the multi-threaded programs we verified.

All in all, verifying that the worker task API was used correctly required 36 annotations in approximately 9,000 lines of code for an average density of one annotation per 248 lines of code.

### Required Modifications

As before, our goal with the JSpider case study was to modify the program itself as little as possible. We made just one modification. Inside the `WorkerThread` class, we commented-out a null assignment. Recall that the worker threads are responsible for executing assigned tasks. As implemented, when a worker thread is assigned a task, it executes the three steps of the worker task protocol and then overwrites its `task` field, where the assigned task is stored, with the null value. The worker thread also maintains a state invariant, that there must be a unique permission to the `task` field. Unfortunately, due to a bug in merging permissions inside of a loop, the null value assigned to the field conflicts with the unique permission expected to be held at the top of the loop. So, we were required to remove the null assignment, as seen on line 28 of Figure 6.9. Note that the commenting-out of this line does not affect the run-time behavior of the program in any important way, with the possible exception of changing what can be garbage-collected.

### True and False Positives

During the JSpider case study, Sync-or-Swim found two data races and reported four false positives. The two data races were found in the `WorkerThread` class. As previously mentioned, the worker thread has an invariant that whenever the `assigned` field is true, the thread's current task must be in the "Init" state. It just so happens that there are two data races on the `assigned` field. We discovered the races because we were forced to annotate the worker thread's methods with share permission, since it is accessed from multiple threads. Two methods access the `assigned` field without synchronization. The field access forces an object unpack, but because the permission is share, Sync-or-Swim issued a warning.

The four false positives all occurred because of one feature that Sync-or-Swim lacks: the ability to specify the elements of an array as holding permission. The worker thread pool uses an array of `WorkerThread` objects in order to keep track of the threads. Each time it assigns a

```
1  @Invariants(@State(name="alive",
2    inv="assigned == true => unique(task) in Init"))
3  class WorkerThread extends Thread {
4    /** Whether this instance is assigned a task. */
5    protected boolean assigned;
6    // ...
7    @Share(use=Use.FIELDS)
8    public void assign(
9       @Unique(requires="Init",returned=false) WorkerTask task) { ...
10     this.task = task;
11     assigned = true;
12   }
13
14   @Share(use=Use.FIELDS)
15   public void run() {
16     // ...
17     while (running) {
18       if (assigned) {
19          task.prepare();
20          try {
21            task.execute();
22            task.tearDown();
23          } catch (Exception e) {
24            log.fatal("PANIC! Task " + task + " threw an excpetion!", e);
25            System.exit(1);
26          }
27          assigned = false;
28          // task = null; Removed due to bug
29       }
30       else { ... }
31     }
32   }
33 }
```

Figure 6.9: The worker thread uses the `assigned` field to indicate whether or not the task is in the "Init" state. (This code excerpt has been edited for space.)

new task to a thread, it searches through the array for an available thread, and calls the `assign` method on that thread. This method requires share permission, but because of the inability of Sync-or-Swim to associate the array elements with permission, this call fails to verify. We should point out that for all of the "symmetric" permissions, there is not much conceptual difficulty to add this feature to Sync-or-Swim, but it is currently not supported.

The four false positives in a program of about 9,000 lines gives us a false positive rate of one per 2238 lines.

| Program | LOC | Protocol | Annotations | | | Warnings | False Positives | Sync. Added |
|---|---|---|---|---|---|---|---|---|
| | | | Inv. | Method | Other | | | |
| Blocking_queue | 107 | open/closed | 4 | 21 | 0 | 0 | 0 | 0 |
| JGroups | 275 | open/closed | 0 | 20 | 3 | 2 | 2 | 0 |
| ECL | 69 | Timer | 0 | 6 | 0 | 0 | 0 | 0 |
| sheltermanager | 1,704 | Timer | 0 | 5 | 1 | 0 | 0 | 0 |
| Twine | 311 | Timer | 2 | 5 | 2 | 1 | 0 | 0 |
| Votebox | 53 | Timer | 0 | 1 | 0 | 0 | 0 | 0 |
| **Total** | 2,519 | | 6 | 58 | 6 | 3 | 2 | 0 |

Table 6.4: Results from a few smaller case studies

## 6.3 Smaller Case Studies

Finally, for the sake of completeness, we also present the results for a number of smaller case studies that were completed in the early phases of our work. The complete results are shown in Table 6.4. Each case study is a small part of a larger open-source program. The size listed in the table (LOC) is the size of the class or classes we verified, rather than of the entire program.

Here is a quick description of each study: Blocking_queue is a thread-shared queue meant to be accessed by one writer thread and multiple reader threads. It defines a simple open/closed protocol, described in earlier chapters of this thesis. The writer can change the state and the reader must not attempt to dequeue items if the queue is not open. JGroups is a middleware for writing distributed communication applications. We verified correct use of the `Channel` class, which defines the principal abstraction of the library, by a demo class included with the program. Both Blocking_queue and JGroups are NIMBY case studies, meaning that they use the atomic block as the principal means of synchronization.

Each of the remaining programs made use of the timer API from the Java standard library, and we attempted to verify its correct use. ECL is a middleware for writing distributed applications based on the ECL protocol. Sheltermanager is an open-source application for managing animal shelters. Twine is another distributed middleware designed to allow programs to discover resources on a heterogeneous network. Votebox is an open-source implementation of software for voting machines.

The `Blocking_queue` method was written without a method for dynamically checking whether or not the queue is closed. We added one so that more interesting client programs could be checked. Additionally, Sync-or-Swim correctly detected a data race in the Twine application.

With 70 annotations in 2,500 lines of code, the result is an annotation rate of one annotation per 36 lines. Two false positives in 2,500 lines of code gives us a rate of about one false positive per 1,259 lines.

## 6.4 Lessons Learned

The overall conclusion we reached is that our approach works reasonably well for specifying and verifying protocols in concurrent programs, and imposes a low burden on the programmer. Our

false positive rates and annotation rates were very similar to that of the single-threaded Plural checker [13, p.112]. Our false positive rates ranged from 1 per 1,259 lines to 1 per 2238 lines. In Plural, they ranged from 1 per 400 lines to 1 per 13,000 lines. Our annotation rates ranged from 1 per 36 lines to 1 per 248 lines. In Plural, they ranged from 1 per 30 lines to 1 per 2,600 lines. Critically, these numbers are comparing a single-threaded analysis to a concurrent analysis.

Still, we learned a number of lessons about what makes a good analysis and what makes the specification process difficult.

### Fields Add Complexity

The more we were required to specify fields with permissions and bring them into the verification process, the more challenging and time-consuming the overall process would become. This statement should not be especially surprising, since the verification of fields has always been one of the more difficult aspects of object-oriented verification. Still, this difficulty came out in certain ways when using Sync-or-Swim.

As we mentioned in the JabRef case study, the verification of the `DefaultMutableTreeNode` protocol was made painful because the class itself and its transitive clients were frequently referenced as fields. Every time the methods called on a field required permission it forced us to create a state invariant and add specifications to the constructor and the method that makes calls on the field. If that class is itself used as a field, the specification process can repeat over and over again. This was especially annoying as a user because the specifications we were adding were very simple: either a share permission, which can be freely aliased, or a share permission with a state guarantee.

In fact, in our experience it seemed that the less important a specification was to verify, the more likely it was to be painful to verify. We believe that there is a certain logic to this. If a type defines a protocol that is important to obey it will often be used in a simple manner, such as locally to a single method body or inside of a wrapper class that defines the exact same states. But for types where the instances rarely or never change states, it is often taken for granted they are in a particular state, and those objects are freely aliased and shared amongst threads. It also makes it all the more frustrating to users since the return on investment, the number of bugs prevented or caught for the amount of time spent, is low.

**Suggestion: Reasonable Defaults or Inference** Many of our issues could have been resolved by either using simple specification defaults or by providing global permission inference. Currently Sync-or-Swim does not have a default annotation for references. Or, to be more accurate, the default annotation is no permission. This is nice when the API you wish to verify is used in a small part of the overall application. It means that most of the program will have no specification and will not add any extra time to the analysis. But, when permissions are pervasive, having a default annotation, notably the share permission for pre- and post-conditions and all fields, would be really helpful. It could save programmers from the seemingly endless cycle of specification required when many fields need permission annotations. The other alternative would be some sort of specification inference, likely global. Specification inference, such as our own inference system presented in Chapter 7, could reduce specification time without giving up on the soundness guarantees provided by Sync-or-Swim.

### Unnecessary Synchronization

Regarding the forced addition of unnecessary thread synchronization, Sync-or-Swim generally did quite well. In all of the JSpider case study, and for most APIs within the JabRef case study, we were not required to add any unnecessary synchronization. But in some cases we were. And in particular, during the verification of the mutable tree API in the JabRef case study, we were required to add a fair number of additional, unnecessary synchronized blocks. Why was this the case? It largely was a symptom of the previous issue, the prevalence of fields that required permission annotation. The more fields that require permission, the more annotation in general that must be written, and in turn the more share permissions that are used. share permissions are the most convenient permission to use because they usually cannot be wrong. They grant the power to modify and can be split freely, so using share permissions means never having to "back-track" to alter a previously written specification.

But at the same time, whenever a specified field is accessed on a receiver that has share permission, Sync-or-Swim requires synchronization. This very general rule protects the state of the potentially thread-shared object when it is accessed by multiple threads. But if the object is not shared, now we have a case where the ease of specification is in direct conflict with our desire to use as little synchronization as is necessary for correctness.

**Suggestion: Special-Case Unique and Full Fields** Based on our experience, we believe a small change in the typechecking rules may lead to a great deal less unnecessary synchronization, while still enforcing typestate invariants, at a cost of missing some data races that are currently detected. As it currently stands, any time an object is unpacked and that object is associated with share, full or pure permission, a lock must be held for that object. This rule is is simple and intuitive. But this rule forces programmers to use extra synchronization in a very common case. That common case is when a programmer unpacks a share object (or full or pure) for the sole purpose of calling a method on a field of share(or pure) permission. We claim that in such a case, synchronization of the outer object need not be required. First, synchronization is not needed to protect the state of the outer object. This is true because state invariants over share and pure fields cannot mention assumed states, and therefore whatever happens to the field between packing and unpacking will be enough to satisfy the state to which the outer object will be packed, regardless of any concurrent access to the field. Next, synchronization is not needed on the outer object to protect the state of the field from data races. It would be if the field were of unique or full permission, but for a share or pure field, if the calling method accesses its state, will be required by our packing rules to acquire its own lock. Therefore it does not need protection from the outer object's lock. Relaxing the rules so that the aforementioned common case does not require holding a lock to unpack would reduce the amount of unnecessary synchronization required.

On the downside, this change would make Sync-or-Swim a worse tool for detecting data races, since field assignments and reads would no longer be required to be protected. Still, given that there are so many other good data race detection tools, it seems like this suggestion might be a wise one to enact.

### Packing Before Calls

In Sync-or-Swim, all objects must be packed before any method call is made, but this can make verification more complicated and is often unnecessary. The motivation for packing objects before method calls is that it always leaves objects in a consistent state, so that if the method indirectly calls back into the object, we know that at least one of its state invariants holds. In Sync-or-Swim packing before method calls is always required except in two cases. The first case is when the method receiver is associated with a unique permission that is held by the calling context for the duration of the call. Since we know that the only permission to the object is held locally, we know that reentrant calls are impossible and so having the object in a consistent state is unnecessary. The second case is when the `@NonReentrant` annotation is used on the class. This unchecked annotation, added in response to the burden we encountered preparing objects for reentrant calls, amounts to a programmer's assertion that no calls in the method will come back into the object.

In general, having objects always be packed is a good thing. This feature gives pure permissions much of their power, since we are guaranteed that even a read-only permission gets to see the object in a consistent state. This feature differentiates our approach from others like Universe Types [35] where read-only references do not get consistent views of the object. Many of the examples we have encountered that involve dynamic state tests would not be possible unless the pure permission was guaranteed a consistent view of the object.

```
@Invariants(@State(name="alive",inv="unique(field)"))
class C {
  Object field;

  @Share(use=Use.FIELDS)
  void foo() {
    C.bar(this.field);
  }

  static void bar(@Unique Object o) { ... }
}
```

Figure 6.10: The receiver must be packed before the call to `bar`, but it cannot be since no unique permission to `field` is available.

That being said, preparing objects for calls by packing them to a state can be challenging. The problem is that the state invariant for at least one state must be satisfied, and in a very commonly occurring pattern, this is impossible. Consider short example in Figure 6.10. The state invariant requires a unique permission to the field `field` in the "alive" state. In the `foo` method, the unique permission is needed to satisfy the pre-condition of the `bar` method, but once taken the receiver cannot be packed to any state. We found that similar patterns are quite common. In our experience programmers usually design their applications so that reentrancy never occurs, so our only recourse was to use the unchecked, `@NonReentrant` annotation.

**Suggestion: Complete @NonReentrant Annotation** The best suggestion we can provide here is to make `@NonReentrant` a first-class part of the approach, rather than an unchecked add-on to the implementation. The need to signify that some calls will not re-enter the calling object is real. We should add such a specification to the theory and design the type system so that it can be checked. This way we will maintain the guarantees provided by Sync-or-Swim but be able to adapt to and verify a very commonly-occurring pattern.

## 6.5 Related Work

A few similar case studies, ones in which analysis tools were used to check API protocol conformance, have been performed in the past and are worthy of discussion.

This author participated in the study that is the most similar to this one [16]. That study was designed to test the ability of the Plural tool, the single-threaded object protocol checker upon which Sync-or-Swim is based, to specify and verify protocol use and implementation. In order to evaluate Plural, we found a few popular APIs in the Java standard library, notably the iterator API and the JDBC database access API, and verified their correct use in a number of programs. Ease of specification, programmer burden and rate of false positives were all noted. The design of our study was based on the Plural study, and in a sense, continues this earlier work. We evaluated more lines of code (roughly 80,000 LOC versus roughly 32,000) and we looked at checking concurrent programs. When compared with the Sync-or-Swim experiments, the Plural study had more focus on specification. A significant portion of the study was devoted to analyzing the expressiveness of the specifications, and in particular the features of Bierhoff's work, dimensions and state hierarchies, which make specification more natural. The Sync-or-Swim study focused much more on verification. Two aspects in particular, share permissions and state invariant verification, received much more attention in our study that in the Plural study. And as a result, we learned that they can be quite difficult in some situations.

Joshi and Sen [67] performed evaluated their dynamic, concurrent typestate checker on a suite of seven benchmarks, including JSpider. Their approach, because it is dynamic, can determine automatically the typestate properties that should be preserved, and therefore their study was also checking the worker task protocol that we checked, and possibly other protocols as well. Like us, they did not find any typestate violations, but they also did not find any data races, where we found two. In the study, their tool reported seven false positives compared to the four false positives reported by our tool. Their study also appears to have examined a good deal more lines of code (643,942 LOC), although there are some discrepancies, since we report JSpider as being a 9,000 line program, and they report it as a being a 65,000 line program.

Naeem and Lhotak [76] performed a comparable study during the evaluation of their whole-program single-threaded typestate checker. They checked correct use of a number of protocols, including the iterator protocol and input and output streams. The test programs consisted of six programs from the DaCapo benchmark suite [18].

Weimer and Necula [96] automatically examined over four million lines of lines of single-threaded Java source in an attempt to find violations of a handful of known object protocols, specifically protocols in which some resource must eventually be disposed. By using a global analysis, they were able to analyze quite a bit more code, since they were not required to write

specifications. They were able to find over 800 protocol violations.

## 6.6 Conclusion

In this chapter we presented a series of case studies designed to evaluate the strengths of our approach in several different areas. We use the Sync-or-Swim tool to specify and verify protocol-defining APIs in two medium-sized programs, JabRef and JSpider, and in several smaller programs. We evaluated our approach for expressiveness of specification, ease of specification and rate of false positives. We found that Sync-or-Swim is able to verify real programs and find real bugs with a reasonable burden and low rate of false positives. We also described our experiences using the tool and made a number of suggestions for improving it in the future.

# Chapter 7

# Probabilistic Permission Inference

*If your desires are not extravagant they will be granted.*

---

This chapter describes Anek, a tool that uses probabilistic inference to infer the access permission annotations described in previous chapters. Given an API annotated with object abstract states and aliasing permissions, Anek will infer the annotations on client code necessary for static verification of the API usage. During inference, Anek solves a series of probabilistic constraints encoding both the logical rules which determine how permissions can be used, as well as various heuristics which encode the most common or "best" specifications in various scenarios. By using probabilistic inference instead of a more traditional inference, Anek is both more efficient and robust in the face of program bugs and analysis imprecision. The following sections describe the tool and its motivation in more detail.

## 7.1 Goals for Anek

While the Sync-or-Swim approach is quite powerful, it has one major drawback; it requires programmers to write annotations at method boundaries. This has the nice benefit of enabling modular checking, but places an additional burden on the programmer who merely wants to ensure correct protocol usage. Therefore, Anek has been designed to eliminate this burden by statically inferring the Access Permission annotations.

We envision programmers using Anek and Sync-or-Swim in the following manner: First, developers of libraries and frameworks would continue to provide Sync-or-Swim annotations along with their APIs. This allows the most knowledgeable developers to build the abstractions specific to their APIs, and formally define the ways in which they must be used. Since an API is typically used by many client programs, this effort is amortized over all the users of the API. When a client wishes to use an annotated API, however, he starts by running the Anek inference tool over his code. Anek will see which annotated API methods are being used and will infer appropriate Sync-or-Swim specifications in the client's code. These annotations become part of the user's code providing documentation for future developers. With the new annotations, the

programmer will then run Sync-or-Swim. Since Sync-or-Swim is a sound checker, if Sync-or-Swim passes the resulting program with the newly inferred annotations, it constitutes a guarantee that the programmer is using the API correctly, with little additional burden on the programmers part. Because Anek performs probabilistic inference based on some heuristics, its annotations are occasionally incorrect. But by running Sync-or-Swim a programmer still gets sound guarantees.

Anek performs probabilistic inference. In other words, it takes a number of constraints based on what specifications are *likely* to be used (rather than what specifications *must* be used) and its solves them in order to determine the most likely specification. We have chosen to develop a probabilistic inference, as opposed to a more traditional, logical inference, for a number of reasons. Essentially, probabilistic inference allows us to encode what is now several years of experience writing Sync-or-Swim specifications. We are encoding the specifications that are most commonly used in different situations. It really means that we can encode intuitions about programs rather than invariants, which must always be true. This is good because it means that the specifications generated by Anek will be idiomatic. In other words, Anek does not just generate *some* legal specification, but rather the legal specification that is likely to be the most desirable according to our own experience. Those familiar with more traditional type inference may wonder if our idea of a "best" type corresponds to the notion of a "principal type [29]." In short, the answer is no. Types in our language cannot be considered to have principal types since there are many occasions when a reference might legally be given one of two permissions. For example, in some program a constructor for a given type might reasonably return unique or full, if the guarantees given by the unique permission kind are never needed. Our notions of good specifications simply come from our own experiences, and are encoded as probabilities. In our example, unique would be better specification, absent all other information, because as the program evolves over time the extra strength of the specification may end up being useful. Unfortunately, unlike type inference strategies like Hinley-Milner [29], our approach is neither sound nor complete. Our overall approach depends on the soundness guarantees provided by the Sync-or-Swim permission checker to ensure safety.

Additionally, because our inference is a known to be approximate, we can build a more lightweight abstraction and perform inference more efficiently than a sound and complete analysis. But perhaps more interestingly, probabilistic inference can infer specifications in situations where a logical algorithm could not. Consider programs containing bugs. Such bugs might very well create unresolvable contradictions in traditional inference algorithms. With probabilistic inference, a "reasonable" specification can still be generated. The same goes for analysis imprecision. Using a probabilistic inference we can still infer specifications even when verifying a specification would fail due to an underlying imprecision in the analysis.

## 7.2 Approach

So, given these goals, the approach taken by Anek is to generate constraints that are likely to be true, based on both heuristics and the logic of permissions, and to solve for those constraints. A solution to the constraints will imply a particular specification for the original program. The algorithm starts by extracting an abstract representation of the program over which the constraints will be generated.

### 7.2.1 Representation

Rather than working on directly on the source code itself, our inference algorithm performs inference over an abstraction of the program. This abstraction is, roughly, a directed graph of the flow of permissions in each program method. While the graph itself represents permission flow, it will as a byproduct encode certain aspects of program control flow. An independent graph will be generated for each method and, later, probabilistic constraints connecting method argument and parameter nodes will be added that essentially turn the inference problem into a global analysis.

The representation is generated in the following manner. For each method parameter, including the receiver for instance methods, two nodes are created. One represents the permission required at the pre-condition and the other represents the permission returned at the post-condition. Using a control-flow graph and a *local* must-alias alias analysis, the permission associated with the incoming object is tracked from the beginning of the method to the end, and various nodes are created in the graph when the object is passed to a method call or used for a field load or store. If the parameter is passed as an argument to a method, then the graph will create a directed edge from the previous node in the graph to a node representing the pre-condition of the corresponding method argument and call-site. The entire process is best explained through a representative example.

```
static void foo(Object x, Object y) {
  bar(x); // call site 0
  bar(y); // 1

  while(*)
    bar(x); // 2

  bar(x); // 3
  x=y;
  bar(x); // 4
}
```

Figure 7.1: A simple method whose representation is given in Figure 7.2

Consider the method `foo` shown in Figure 7.1. This method gives rise to the abstract representation shown in Figure 7.2. There are several interesting features worth pointing out. First, look at the permission that flows from the pre-condition for the parameter `y` to the post-condition for the parameter `y`. At each call to the method bar there are a pair of nodes, one for the pre-condition of the 0th argument and and one for the post-condition of the 0th argument. Each call site is given an separate identifier, here S0 through S4, corresponding to the five calls to bar in Figure 7.1.[1] The pattern for each call site is similar. Some permission goes into the method, as specified by its pre-condition. No permission comes out of the pre-condition node, but some may come from the post-condition node. Additionally, some permission may be 'held' by the calling

[1]Because Anek tracks each call-site separately, it is in some sense context-sensitive. This was done in the anticipation that it might eventually be useful, although at present the implementation does not take any specific advantage of this feature.

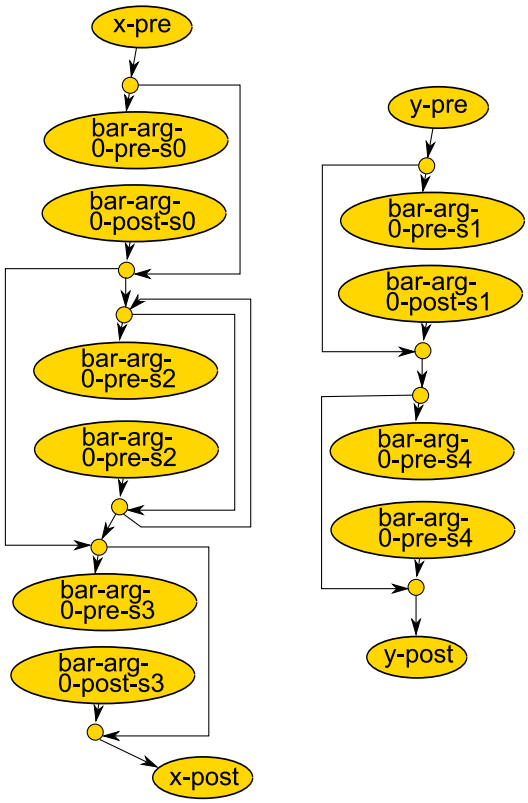


Figure 7.2: The representation generated for the method in Figure 7.1

context, for example, when the permission required by a call is weaker than the one held in the calling context. Therefore, a separate edge is generated to represent this permission.

As previously mentioned, the representation is generated with the help of a local must-alias alias analysis. At the beginning of the method body each parameter is associated with an abstract object location. If the alias analysis is sure that location is aliased by another reference, it will treat all actions on the other reference as actions on the original permission. In this example that means that the call to bar on the last line of the program (call site four) is treated as if `y` were passed as the argument, since the call is definitely on the object that `y` pointed to when the method was called.

The path followed by the permission to the `x` parameter is similarly interesting because of the 'while' loop. Thanks to the loop, the graph has edges representing the flow of permission for the case where the loop is never taken and for the case where the loop goes back around.

In the case where the method is an instance method, the flow of permission to the receiver parameter is represented in the same manner. Our graph also represents method returns, both of the methods under analysis and of called methods, and field loads and stores. Method returns of called methods are represented by a node in the graph. This node will generally act as a permission source, since it will have no incoming edges, but its permission may flow into other nodes (assuming the return value is assigned). In the case that the same method is called multiple times, each call site return is represented by a different node. Each method under analysis has

one node representing the permission returned from that method (assuming it has a non-void return type). This node will act as a permission sink, with only incoming edges.

Once the representation is generated, we will associate random variables with each node and edge in the graph.

### 7.2.2 Random Variables

Given an abstract program representation, Anek associates each node and each edge between nodes with a series of random (i.e. *probabilistic*) variables. A probabilistic variable is a Boolean variable which has a certain probability of being true, and a certain probability of being false. Each node has one random Boolean variable for each of the five permission kinds,

$$v_{unique}, v_{full}, v_{immutable}, v_{share}, v_{pure}.$$

Additionally, for each abstract state in the hierarchy of the type with which the node is associated, that node is given a random Boolean variable. So, for a node associated with a parameter $x$ where the parameter is of file type, that node would have two random variables $v_{open}, v_{closed}$, one for each of its abstract states.

Each random variable will be given a *prior distribution*. This is the likelihood that the variable is true before any other constraints are added. For most of the variables Anek creates, we have no idea if they should be true or not, since we are trying to infer the specifications. For that reason, most variables are given a prior distribution representing a 50% chance of being true.

However, if a specification already exists in the source program, we consider this to be a very good indication as to the probabilities of the random variables, and we adjust our prior probabilities accordingly. For example, consider the following specification for the receiver parameter:

```
@Perm(requires="full(this) in Open",
      ensures="full(this) in Open")
byte[] read() {...}
```

Based on this specification Anek will modify the prior distributions for the receiver pre- and post-condition nodes. For the pre-condition node, it will set the prior probability for both the full permission kind and the open abstract state to a high probability (99%). The remaining variables will be given a low prior probability (1%). Therefore, the prior probabilities for each of the random variables will be as given in the following chart:

| **Random Boolean Variable** | $PriorProbability$ |
|---|---|
| $v_{unique}$ | 1% |
| $v_{full}$ | 99% |
| $v_{immutable}$ | 1% |
| $v_{share}$ | 1% |
| $v_{pure}$ | 1% |
| $v_{open}$ | 99% |
| $v_{closed}$ | 1% |

Note that even though the specification is given, we still say that the specification permission is only very likely to be true (i.e., true with 99% probability) not definitely true (i.e., true with

100% probability). This allows for the possibility that the original specification was incorrect if the evidence against it is overwhelming.

Each edge in the graph connecting two nodes is also given the same series of probabilistic variables. This allows probabilities of one node to influence the probabilities of adjacent nodes, as discussed in the next section.

Once the prior probabilities are set for each of the random variables, we further constrain the variables based on the shape of the graph and the properties of the nodes it contains. These constraints will further modify the likelihood of each random variable being true, and after the constraints are solved Anek will find the specification corresponding to the most likely random variables.

#### Likely Permission Kinds

While the proceding discussion gives an intuitive idea about how Anek works, in practice we found that setting each permission kind for each unspecified node with a prior probability of 50% resulted in unsatisfactory results. In practice, certain permission kinds are more common in specifications than others, and so we changed our prior probabilities to reflect this. full and share permissions are the most common permission kinds in our experience, followed by the unique permission kind and followed lastly by the pure and immutable permission kinds. This observation was reflected in our implementation in the following manner:

First, for normal, unspecified nodes and edges, unique, full and share permission kinds are given a 50% prior probability, while immutable and pure are given a 40% prior probability. Then, if the nodes are "on the path" of a specified permission, we further bump up their prior probabilities. A node is on the path of a specified permission if a path exists from that node to a node that was given a concrete specification in the original program under inference. Because program graphs are method-local, the scope of this effect will only restricted to a single method. We did this after our early experiments indicated that, after inference, every node was either much too likely to have a specification or much too unlikely. In reality, we wanted to mostly ignore nodes that were not involved with the API being checked, and be more likely to infer a specification for nodes that were involved with the API being checked. This extra increase was a way to accomplish just that. So, for nodes determined to be on the path of a concrete specification, the prior probabilities for full and share permission kinds are 70%, 60% for unique and 50% for immutable and pure.

All of the prior probabilities were determined from experience on Plural and Sync-or-Swim case studies, as well as being influenced by the evaluation programs described in this chapter. We expect that over time the relative likelihood of each of the permission kinds will change to reflect our better understanding.

### 7.2.3 Constraints

Setting the prior probabilities for the random variables in our graph gives us a starting point, but we must add additional constraints in order to encode our heuristics and based on the features of the program itself. Once these constraints are added, we will solved for the random variables and this solution will give us our specification. Specifically, for each node in the program we

will find out which permission kind and which abstract state are most likely to be true. If the likelihood of the most likely permission kind and state exceed some threshold, then the pair will be chosen as the specification.

But what exactly are the constraints that we add? We have broken down the constraints on random variables into two classes, heuristic and logical. For our purposes, heuristic constraints encode features that are *generally* true of good Sync-or-Swim specifications. Logical constraints, on the other hand, merely encode the basic permission rules, things like the rules governing sound permission splitting. Importantly, even though the logical rules must always be true in a program verified by Sync-or-Swim, Anek only dictates that they be true with some high probability. It is precisely this feature which allows Anek to infer specifications even in the face of buggy programs and imprecise features of the static analysis. In the next two sections we will present each of the constraint generations rules in turn.

For all of the constraints described in this section, some relationship will be generated that must be true with either high probability (W.H.P.) or with low probability (W.L.P). The concrete percentages that were chosen to represent either high or low probability have been parameterized, and are chosen by the user of Anek so that he may tune the performance of the algorithm. In our experiments, high probability was set to be 90% for the logical constraints and 70%-80% for the heuristic constraints. Low probability was set to be 10% for the logical constraints and 20%-30% for the heuristic constraints.

### Logical Constraints

Anek encodes the basic logic of Access Permissions through a series of logical constraints. While in reality each of the constraints must be satisfied for a program to be verified with Sync-or-Swim, Anek only says that they must be true with high probability.

**Field Write** For any field store node (i.e., field assignment), the associated receiver node is almost certainly not associated with one of the read-only permissions, immutable or pure. This constraint then sets the receiver to be immutable or pure with a very low probability. A field cannot be modified without writing permission to its receiver, so whenever we see a field store, we can assume that we have writing permission to the receiver object. We constrain the reading permissions to be unlikely (as opposed to making the writing permissions likely) because we want to minimize the chance that a specification is inferred where *no* permission was actually necessary.

**Outgoing Permissions** At any nodes with multiple outgoing edges, constraints must be generated which relate the permission at the node to the permissions on the edges. In some cases, nodes have multiple edges because the permission at the node is being split into multiple permissions. In other cases, the multiple edges is an effect of control flow branches. Since permission splits can only occur before method calls and field reads and return nodes, we can mark these nodes as such, and apply different rules for splitting and control flow branches. At branches, we simply apply the rule that the permission available at the node is equal to the permission available at each of the outgoing edges. For permission splits, however, the permission on the outgoing edges generally cannot be identical.

There are certain ways in which Access Permission can be soundly split. For example, a unique permission can be split into two share permissions, two immutable permissions or two pure permissions. It cannot, however, be split into two full permissions or two unique permissions, as those two newly created permissions would violate the assumptions made by one another. Therefore, at each node, a number of constraints are placed on the edges leaving the node. If only one edge is leaving the node, then Anek says that the permission on the node and the permission on the edge are the same with high probability. If two edges leave the node then Anek then we add equality constraints on the random permission kind variables as follows.

The likelihood that each outgoing edge is a share permission is constraint to be equal to the likelihood that the node itself is share with high probability:

$$v^n_{share} = v^{e_1}_{share} = v^{e_2}_{share} \text{ W.H.P.}$$

The same is true of the immutable variables:

$$v^n_{imm} = v^{e_1}_{imm} = v^{e_2}_{imm} \text{ W.H.P.}$$

The likelihood that the node itself has unique permission is constrained to be equal to the likelihood that the first edge is unique or the likelihood that the second edge is unique. Furthermore, it is unlikely that both are unique:

$$v^n_{unique} = v^{e_1}_{unique} \vee v^n_{unique} = v^{e_2}_{unique} \text{ W.H.P.}$$
$$v^{e_1}_{unique} \wedge v^{e_2}_{unique} \text{ W.L.P.}$$

Finally, there are constraints describing the means by which full permissions can be split. The likelihood that the node has full permission is constrained to be equal to the likelihood that one edge is full and the other pure or the reverse. And again, the likelihood that both edges are full is low:

$$v^n_{full} = v^{e_1}_{full} = v^{e_2}_{pure} \vee v^n_{full} = v^{e_1}_{pure} = v^{e_2}_{full} \text{ W.H.P.}$$
$$v^{e_1}_{full} \wedge v^{e_2}_{full} \text{ W.L.P.}$$

One may notice that a few splitting scenarios have not been covered. For example, the possibility of a unique or full permission splitting into two share(s), two immutable(s) or two pure(s). In practice, adding these splitting rules did more hard than good by making it more likely for unnecessary specifications to be inferred. Furthermore, they did not actually seem to be necessary since when they did come up other constraints would make up for their absence. Still, the omission of these constraints amounts to a heuristic the appropriateness of which needs to be fully evaluated.

Graphs where a single node has more than two edges leaving it will only be generated as a result of control flow jumps to multiple locations (i.e., not as a result of permission splits). Therefore, in such cases we constrain all outgoing edges to be equal to the permission at the node.

**Incoming Permissions** When a node has incoming edges, Anek also adds constraints on the relationship between the incoming edge permissions and the node permission. Specifically, Anek says that the permission associated with a node is equal to one of the permissions on the incoming edges with high probability. This case does not apply when the node is the merge node generated after method calls.

**Borrowing** Borrowing is a feature of Sync-or-Swim that we have not yet covered. Sync-or-Swim allows specifications to be written in a manner such that a method guarantees it will return exactly the same permission fraction it was given. This is known as borrowing, and specifications that borrow permission to their parameters are in general more widely useful because a strong permission (e.g., unique) can be passed to a weaker required permission (e.g., full), and if the permission is borrowed, the strong permission can be reestablished after the method call returns. Because borrowed specifications are so useful, Anek tries to infer them whenever possible. In the path of one parameter permission from pre-condition to post-condition, the specification can be a borrowing specification when there are no paths from the pre-condition that do not end at the post-condition, and if every method called on that permission is also borrowing. Anek explores the path of each parameter permission from pre-condition to post-condition. If the path terminates only at the post-condition, it will constrain the permission to be a borrowing permission with high probability. If the path terminates only at a post-condition but passes through several method calls along the way, then Anek generates a constraint saying that the specification is likely to be borrowed if each of the called methods borrow their permissions. Otherwise, Anek will constrain the specification to be a borrowing specification with low probability.

**Method Call** Anek generates very specific constraints at method call sites, constraints that may be different than the constraints normally generated for incoming/outgoing edges on a node (as described above). In particular, if the method call borrows its permission, we want the permission leaving the post-condition to be the same as the permission entering the pre-condition. Therefore, at method call sites, Anek generates a constraint that says if the parameter permission is likely to be borrowed then the permission leaving the call is the same as the permission entering the call with high probability. Otherwise, the permission leaving the call is equal to either the permission returned from the post-condition or the permission that did not go into the pre-condition (the permission that was saved in the calling context) with high probability.

**Interprocedural** Anek adds constraints to say that the permission on each argument node of a method call site is likely to be equal to the permission on the corresponding method parameter for that method with high probability. Adding this constraint effectively turns Anek into a interprocedural analysis since it allows constraints to propagate from one method to a called/calling method.

**Overridden Methods** Because Sync-or-Swim has support for inherited specifications (and because they are needed extensively in our case study program), we wanted Anek to be able to infer specifications which were consistent with Sync-or-Swim's rules. At issue is the specification of overriding methods. The Sync-or-Swim methodology requires that specifications on methods be consistent with any methods they override. Consistent means that they obey behavioral subtyping.

Because of the possibility that an overridden method could be inferred to have a new specification, Anek adds constraints that say, essentially, an overriding method is very likely to have the same specification as the method it overrides. These constraints are also interprocedural.

**Heuristic Constraints**

For the random variables generated from a program representation, a series of additional constraints are added which correspond to our intuitions about what makes a good Sync-or-Swim specification.

**Constructors** Constructors generally return unique permission, so for the specification for the object created by a constructor, we say that the variable $v_{unique}$ is likely to be true with elevated probability.

**Pre and Post** For a given parameter of a method, the permission kind, but not the state, of the pre and post condition nodes are generally the same, and the permission is borrowed.

**On Path** As previously mentioned, when setting the prior probabilities, any node in a method graph that is a predecessor of a node that was specified with a concrete permission in the original program under inference will be more likely to have some kind of permission. The prior probabilities for all five permission kinds are elevated for such nodes.

**Create Methods** Methods whose names begin with the word "create" generally return a unique permission, much like a constructor. These methods in practice are often static factory methods.

**Setter Methods** Methods whose names begin with the word "set" generally require a writing permission (i.e., Unique, Full or Share). Therefore, when encountering such a method in the representation, the variables $v_{immutable}$ and $v_{pure}$ for the receiver pre- and post-condition are constrained to be true with low probability.

**Thread-Shared** Targets of synchronized blocks are of full, share or pure permission with high probability. This heuristic is based on ideas developed in the concurrent version of the Sync-or-Swim analysis. Full, share and pure are the three permission kinds that may indicate possible racy thread-sharing.

## 7.3 Architecture

We now give a brief overview of the Architecture of the Anek tool. Anek is essentially divided into five components. Since Sync-or-Swim is an Eclipse plugin, two of the components are needed interface with Eclipse's Java IDE. The rest of the components are used to generate constraints and solve them.

Each component of Anek is essentially organized into a pipeline. The first component, the Graph Extractor, is a plugin to the Eclipse JDT, the Java IDE in the Eclipse. Its job is to visit the Java AST generated by JDT for the program under inference and generate the abstract representation. This representation is stored to disk in an intermediate, XML-based format. This component is also responsible for the user interface, the menus and action items in Eclipse that

allow users of Sync-or-Swim to run inference without ever leaving their IDE. This component is written in Java.

Once Anek has generated an XML representation of the program, this representation is handed off to an F# program. The use of F# is one of necessity. Our constraint solver is Infer.NET, a library on the .NET platform, so the use of some .NET language was required. But additionally, our program representation is abstract enough that it could be generated from programs written in a number of different OO programming languages.

Once the program representation is loaded into memory, the Node Constraint Generator generates a number of constraints based on the shape of the program. These are the constraints described in the previous section. However, at this phase each of the constraints is deterministic; there are no probabilities involved.

It is in the next component, the Probabilistic Constraint Generator, that the list of constraints generated by the previous phase are used to generate a number of corresponding probabilistic constraints on Boolean random variables. This component is also responsible for creating those random variables and setting their prior distributions.

At this point, the constraints are handed off to Infer.NET, which Anek uses as a black box to solve its probabilistic constraints. The Infer.NET framework exports a Variable interface to its clients which defines an abstraction of a Boolean random variable. One nice thing about this abstraction is that many standard Boolean operators, for example logical AND and logical OR, can be applied to these random variables. This makes it rather easy to encode the constraints described in the previous section. To give an example, consider the constraint on unique permissions at a node with multiple outgoing edges, described in the previous section:

$$v^{n}_{unique} = v^{e_1}_{unique} \vee v^{n}_{unique} = v^{e_2}_{unique} \text{ W.H.P.}$$
$$v^{e_1}_{unique} \wedge v^{e_2}_{unique} \text{ W.L.P.}$$

These constraints are encoded as follows:

```
Variable<Boolean> v_c1 = v_n_unique === v_e1_unique;
Variable<Boolean> v_c2 = v_n_unique === v_e2_unique;
Variable.ConstrainEqualRandom( v_c1 ||| v_c2, 0.90 );

Variable.ConstrainEqualRandom( v_e1_unique &&& v_e2_unique, 0.10 );
```

Where the method `ConstrainEqualRandom` makes the Boolean random variable it is passed as the first argument true with the probability given as the second argument.

Based on the random variables that are created and the constraints applied to them, Infer.NET generates a number of factor graphs and solves them using an approximate algorithm such as expectation propagation [74]. With the results returns by Infer.NET, Anek goes through each node in the graph rewriting its permission. If the most likely permission kind and abstract state are more likely than some threshold value, as determined by Infer.NET, the generated graph will contain this newly inferred specification. This newly generated representation, which matches the same XML schema as the original representation is written to a file.

Finally, in the last component, the newly generated representation is loaded from disk and applied to the original Java program inside another Eclipse plugin called the Graph Applier. This program, written in Java, walks through the AST of the original Java program applying the

specification as given by the new representation file. After this process is complete, the inference procedure is done.

## 7.4 Experiments

In order to evaluate the utility of Anek we performed a number of small experiments and one main experiment. The goal was to see if Anek would infer annotations that were correct and would not lead to a large number of warnings when running the Sync-or-Swim tool.

First we developped a number of test-sized experiments. Each of these experiments consisted of one or more classes, with one or more methods, some of which were annotated by us before running the Anek tool. Each experiment was designed to test some particular Anek constraint or feature. During the experiments we would run Anek on the test suite, and ensure that correct annotations were inferred, and that after inference Sync-or-Swim would report no warnings. At issue is the evolution of Anek in response to newly perceived problems. One of the great benefits of Anek's architecture is that it is so easy to evolve it by adding new constraints. Over the course of its implementation we added new constraints or modified existing constraints numerous times. However, we wanted to ensure that the new constraints, which may fix one particular problem, did not come at the expense of any previously-correct behavior. Therefore, our small experiment suite formed a regression suite of sorts.

Our primary experiment, though, was to use Anek to infer annotations for the PMD static analysis framework. In this experiment, the Java Iterator API was annotated and then Anek was used to infer annotations within the PMD application, which makes extensive use of the API.

We had multiple reasons for choosing this particular program as a case study. The API is fairly simple and does not make use of Sync-or-Swim features that Anek does not yet support, for example state dimensions. The program was medium-sized (see Figure 7.4 for details) so we could get a feeling for how Anek scales. Also, iterators are typically used in a fairly straightforward manner, local to one or two methods, and therefore we would not have to worry about inferring state invariants, which Anek currently does not support. However, PMD still contains enough interesting uses of the API, (e.g., subclassing, interprocedural, and higher-order) to make the case study worthwhile. Finally, because the program was used as a case study in earlier work [13], it is a useful point of comparison.

| **PMD** | |
|---|---|
| Lines of Source: | 38,483 |
| Number of Classes: | 463 |
| Number of Methods: | 3,120 |
| Calls to Iterator.next(): | 170 |

Figure 7.3: Simple statistics for the PMD application.

Specifically, PMD was used as the client-side case study in Kevin Bierhoff's doctoral thesis. In his experiment, the author took an annotated Iterator API, ran Sync-or-Swim on PMD, and added appropriate annotations *by hand* to the program until there were as few remaining warnings

as possible. Our goal was essentially to replicate this experiment by using Anek instead of doing any specification by hand.

| Method | Annotations | Warnings | Time Taken |
|---|---|---|---|
| Original | 0 | 45 | 0 |
| Bierhoff | 26 | 3 | 75min [13] |
| Anek | 32 | 4 | 7min 31sec |
| Anek ("Logical") | 27 | 43 | 7min 55sec |

Table 7.1: The results of running Anek on PMD.

Figure 7.4 contains a number of basic statistics for the PMD application. Of particular note is the number of calls to the method, `java.util.Iterator.next`. This is important because the `next` method is the most important for verification purposes. It is the only method on the Iterator interface that requires the iterator instance to be in a particular state when called.

Table 7.4 shows the results of our experiments. We ran four experiments and recorded three statistics. The first configuration is "Original." In this experiment, we ran Sync-or-Swim on PMD with no annotations at all, in its original form. The point of this experiment is just to show that *some* work must be done in order to verify PMD's use of the Iterator API. And to that end, Sync-or-Swim reported 45 warnings when run on the unannotated program. The next configuration, "Bierhoff," is PMD as annotated by Kevin Bierhoff for his thesis work. Manual annotation took 75 minutes as reported by the author. Sync-or-Swim reported three warnings. In all three cases, the warnings were false positives. In these three cases, the `next` is called on an iterator without first calling the `hasNext` method to establish dynamically that the iterator has subsequent elements. In all three cases, other program invariants not expressed in Sync-or-Swim guarantee that the call to `next` will not fail at run-time because the underlying collection is known to be non-empty.

The next two experiments use Anek to infer annotations on PMD. The "Anek" configuration is the standard configuration. When running Sync-or-Swim on PMD with the annotations inferred by Anek, four warnings are generated, and the inference process takes 7 minutes and 31 seconds. Of the four warnings, three are the same warnings issued by Sync-or-Swim as in the "Bierhoff" configuration. The fourth warning, which is also a false-positive, can be attributed to Anek not performing branch-sensitive inference. It is our belief that with branch-sensitivity, Anek will do as well as with hand-coded annotations.

Finally, the "Anek Logical" configuration is a rough attempt to simulate a non-probabilistic inference algorithm. In this configuration, all heuristic constraints are turned off, and all probabilities are set to either 1.0 (for true) or 0.0 (for false). While a standard inference algorithm would likely be implemented in a much different manner, using a SAT solver or similar such engine rather than a probabilistic constraint solved, this still gives us some idea of what we might expect. In this configuration, Sync-or-Swim reports 43 warnings, all of which are false-positives.

## 7.5 Discussion

In this section we will discuss the results of our experiment and their ramifications. Overall, we were quite impressed with the results of the experiment. In approximately 10% of the time it took to annotate and the program by hand, Anek was able to infer specifications that were almost as good, and with no human involvement. Specifically, the specifications inferred by Anek lead to four warnings when the Sync-or-Swim tool was subsequently run on the result, versus three warnings from hand-written specifications. This difference of one warning is entirely due to Anek's lack of path-sensitivity in its inference scheme, a feature Sync-or-Swim itself supports. Precision-wise, Anek does a very good job with the annotations it infers.

We also claimed that our approach is a good idea because probabilistic permissions enable an analysis that is more robust to bugs and false-positives. While our experiments leave the question of bugs unresolved, for false-positives the results are good. Three of the four warnings issued by Sync-or-Swim on PMD were due to imprecisions in the underlying analysis. This shows that Anek can be an effective inference tool even when the resulting specifications will not be verifiable.

| **Description** | **Count** |
|---|---|
| Same | 14 |
| Anek Added Helpful Spec. | 6 |
| Anek Added Constraining Spec. | 1 |
| Anek Removed Spec. | 3 |
| Anek Changed Spec., More Restrictive | 6 |
| Anek Changed Spec., Wrong | 3 |

Table 7.2: Comparison of by-hand annotations with Anek

The quality of the specifications inferred by Anek is generally good. Table 7.2 summarizes the annotations inferred. Specifically, these numbers are for the specifications inferred by Anek with respect to the original, hand-specified version. 14 of the specifications were exactly the same. Anek inferred 6 specifications that were correct, potentially useful in future versions of the application and imposed no additional proof burden. In one case Anek added a specification that was not necessary and may, in the future, cause additional proof burdens. In 3 cases Anek did not infer a specification that was present in the hand-specified version. All three of these were related to dynamic state test methods, which Anek currently does not attempt to infer. The removed specifications were immaterial because at all use sites, a super-type specification took precedence. In 6 places, Anek changed an existing specification to make it more restrictive, which, while not causing any additional errors now, may lead to additional proof burdens in future versions of the application. Finally, 3 specifications were wrong outright. One of these incorrect specifications lead to the additional warning. The other two did not affect verification at all.

One nice benefit of creating an analysis based on probabilistic constraints is the ease of design. It turned out to be quite easy to add new constraints. As we went through our design iterations, we started with a basic suite of probabilistic constraints that we thought would yield

good results. However, on some of our small benchmarks we found that for one reason or another these constraints were not quite yielding the expected results. Fortunately, it is quite easy to add new constraints. So, as we realized that one constraint was overly specific, or that another, say, did not work in all situations, it was trivial to add a new constraint so that the results would be more to our liking.

## 7.6 Related Work

Probabilistic inference is not an entirely new area. Our work was directly inspired by Merlin [73], a tool for inferring security annotations. Merlin is designed to infer the "source," "sink," and "sanitizer" annotations useful for performing a static taintedness analysis. The locations of program nodes in a control-flow graph indicate to Merlin which functions are likely to be sources (creating possibly-tainted data), sinks (consuming data that must not be tainted) and sanitizers (turning tainted data into untainted data). After solving probabilistic constraints, Merlin labels the most likely nodes as such. Interestingly, it is nothing about the code in the functions themselves that tell Merlin whether or not that function is, say, a source, it is rather how the function is used in the larger program. In general the specifications that we are trying to infer with Anek contain much more detailed behavioral properties. As such, Anek must know much more about the details of each function's behavior. Additionally, with Anek we have the nice feature that after specifications are inferred, a sound static analysis can be run, verifying the results of the inference and acting as a safety net. For Merlin, such a tool was unavailable.

Kremenek et al. [70] propose a tool for inferring ownership annotations that is also built upon probabilistic analysis. While the specifications they are trying to infer contain less detailed behavioral information, ownership annotations themselves are rather similar to our own access permission annotations.

As part of his Ph.D. thesis, Dietl [34] developed a global analysis for inferring Universe Type annotations. His approach is implemented using a Boolean satisfiability (SAT) solver. The interesting part of this work is that in general there is no "best" solution for ownership type inference. Inference could easily infer a flat object hierarchy which would always be sound, albeit uninteresting. So heuristics are used to weight the graph in order to coerce solutions that are more interesting with respect to object topology. A Max-SAT solver then can be used to find the most interesting satisfiable solutions. Besides the fact that we are inferring typestate annotations, the primary difference between this work and ours is that it still requires satisfiability. If a program has bugs and therefore has no valid ownership type, the inference will fail with unsatisfiable constraints. Ours will always produce the best possible specification.

Terauchi [92] proposed a global analysis for inferring fractional permissions in order to verify a lack of race conditions. While the methodology itself solves a problem that might be useful in our work, their underlying methodology is much different, since they do not use probabilistic constraints. Presumably such an analysis would have to give up when confronted with false posistives of the sort we encountered in our case study.

## 7.7 Conclusion

In this chapter we presented Anek, a probabilistic specification inference tool that can be used to infer access permissions for use with Sync-or-Swim. Anek is novel in that it is built using probabilistic constraints. Those constraints allow us as developers to encode into the analysis our understanding of what makes a good specification. Probabilistic constraints also make Anek robust to bugs in the program under inference or imprecisions in the eventual analysis. In order to evaluate our approach, we used Anek to infer specifications for PMD, mimicking a case study that was performed by Kevin Bierhoff by hand as part of his Ph.D. thesis. The results were good. The specifications inferred by Anek were nearly as good as those written by hand, while still be robust against imprecisions in our protocol-checking approach.

# Chapter 8

# Reducing STM Overhead with Access Permissions

*Pay attention. An opportunity will knock on your door.*

---

In this chapter we show that access permissions, in addition to enabling the static verification of concurrent programs, can enable their optimization. We describe a technique for using access permissions to statically reduce the overhead associated with software transactional memory runtime systems. A previous version of this chapter appeared at the 2009 IWACO workshop [11]. The work itself was done in collaboration with Yoon Phil Kim and Sven Stork, and forms the basis for Yoon Phil Kim's masters thesis [69].

## 8.1 Introduction

Transactional memory [57], or TM, is a promising approach to decreasing the difficulty of writing multi-threaded, shared-memory applications. TM systems provide programmers with a new primitive, the atomic block, whose simple semantics dictates that code inside the block must be run *as if* no other threads were running concurrently (Figure 8.1). This primitive is typically implemented in an optimistic fashion, wherein threads run concurrently but have the effects of their memory writes "un-done" if they were able to observe a view of memory inconsistent with atomic semantics.

Unfortunately, there are some obstacles to the wide-spread adoption of this approach. One obstacle is the relatively large overhead that existing transactional memory systems impose over standard lock-based synchronization. This overhead is primarily due to required instrumentation, as certain *logging* and *synchronization* operations must be performed on every (or at least many) memory accesses.

In this paper we propose an optimization of *software* transactional memory or STM [53]. Our optimization will use the system of access permissions presented in previous chapters. In this optimization, for certain references, such as those that point to immutable objects, we will be able to remove all synchronization and logging operations. For other references, such as those that

```
void xfer(Account a1, Account a2, int amt) {
  atomic: {
    a1.withdraw(amt);
    a2.deposit(amt);
  }
}
```

Figure 8.1: A very simple use of the atomic block in a Java-like language. Note that this snippet uses Java's labeled block in order to maintain compatible syntax, as does our implementation.

uniquely point to the object to which they refer, we will be able to remove all synchronization overhead associated with accessing the object (because the object was in fact thread-local) or we will be able to treat that object as part of the protection domain of another object (because the object was transitively accessible from a thread-shared object). In each case, our optimization reduces the overhead of STM.

This chapter makes the following contributions:

1. We present a technique for the compile-time removal of unnecessary synchronization and logging in STM implementations based on access permissions, an existing alias control mechanism.
2. We have implemented this optimization in AtomicPower, a source-to-source implementation of STM based on AtomJava [62] and work by Adl-Tabatabai et al. [1]. AtomicPower takes a program written in Java extended with atomic blocks and translates it into an optimized Java program.
3. We have evaluated our optimizations on a number of benchmarks, including an open-source video game application. In general performance is improved, and in certain cases greatly improved, ranging from 10% to 40% improvement.

Since access permissions were designed to aid in the verification of behavioral properties of object-oriented programs, we claim that programmers who are already using this system to verify concurrent programs can take advantage of our optimizations without any additional specification burden.

We proceed as follows: Section 8.2 first describes our implementation of software transactional memory and then describes our permission-based optimizations to that implementation. In Section 8.3 we describe our evaluation procedure, our benchmarks, and the results of our optimization. Finally, we discuss related work and conclude.

## 8.2 Approach

We implemented STM as a source-to-source translation, from Java with certain labeled statements delineating atomic blocks (those labeled as `atomic`) to pure Java. We then used static access permission annotations to remove unnecessary synchronization and logging. Before describing our optimization, we briefly discuss our initial implementation of STM in order to show what kinds of synchronization and logging operations are normally necessary. Our optimization

is able to reduce overhead on accesses to immutable and unique references, and to a lesser extent, full references.

### 8.2.1 Base Implementation

Our implementation of software transactional memory is a combination of AtomJava [62] and work by Adl-Tabatabai et al. [1]. AtomJava is a source-to-source implementation of STM that uses a pessimistic synchronization strategy. It takes programs written in "Java plus atomic blocks" and outputs pure Java source code. We used AtomJava as a starting point, but rewrote much of the internals and run-time system in order to use the synchronization strategy proposed by Adl-Tabatabai et al. [1]. While we have attempted to make our implementation as fast as possible, we do not claim excellent absolute performance. Rather, we claim that we can improve relative performance by reducing the number of synchronization and logging operations required. It is our belief that access permissions could help optimize many different implementations of STM, but that the optimization might be slightly different with other design choices.[1]

Our implementation uses an optimistic read, pessimistic write strategy with object granularity. Each object is either owned[2] by a thread inside a transaction, or unowned. Unowned objects can be read at will by any transaction, but in order to write an object, a transaction must be the owner of that object, and it remains the owner until the end of the transaction. Writers modify objects in place, and roll back the state of the object in case of transaction abort. We use a version numbering scheme in order to detect possibly-inconsistent reads.

The source-to-source translation process begins by rewriting every object to (transitively) extend TxnObject which holds a TxnRecord for storing object meta-data. The TxnRecord contains both an owner field, telling transactions whether or not the object is owned and by whom, and a version number. Every thread in the program is rewritten to extend TxnThread. TxnThread itself extends java.lang.Thread, but holds a TxnDescriptor object which contains additional data related to a transaction's status. TxnDescriptor holds three thread-local hash maps, one each for the read set, write set and undo log.

Our implementation must also rewrite atomic blocks and memory reads inside transactions. Like AtomJava, we create two copies of each method, the original version and a version to be called inside of atomic contexts. An atomic block is rewritten as a loop that initially calls `txnStart`, setting the current transaction's status to 'active.' The loop contains a try-finally block whose finally block attempts to commit the transaction, re-executing the loop if the transaction commit fails. Field reads (and writes) in an atomic context are replaced with calls to `txnOpenObjectForRead` (or Write), which obtains the object's TxnRecord and calls `txnOpenRecordForRead` (or Write), whose implementations are shown in Figure 8.2. Note that the `isOwned` method has cost equivalent to a volatile read, and `setOwner` must perform an atomic test-and-set. The `logWriteSet` method performs a whole object copy and a hash table insert, while logReadSet performs a hash table insert.

[1]The AtomicPower implementation is available at `http://www.nelsbeckman.com/research/atomicpower/`.

[2]Unfortunately, the "owned" terminology comes from the STM community. This notion of transaction ownership, which we will use exclusively throughout the remainder of the chapter, is in no way related to Ownership, the alias control mechanism.

```
static void txnOpenTxnRecordForRead(TxnRecord rec) {
  TxnDescriptor txnDesc = getCurrentThreadTxnDescriptor();
  if ( txnDesc.writeSetContains(rec) )
    return;
  do {
    if (!rec.isOwned()) {
      logReadSet(rec, txnDesc);
      return;
    }
    txnHandleContention(rec);
  } while (true);
}

static void txnOpenTxnRecordForWrite(LoggableObject obj,
    TxnRecord rec) {
  TxnDescriptor txnDesc = getCurrentThreadTxnDescriptor();
  if ( txnDesc.writeSetContains(rec) )
    return;
  do {
    if (!rec.isOwned()) {
      if (rec.setOwner(null, txnDesc)) {
        logWriteSet(obj, rec, txnDesc);
        return;
      }
    }
    txnHandleContention(rec);
  } while (true);
}
```

Figure 8.2: The implementation of the methods txnOpenTxnRecordForRead() and txnOpenTxnRecordForWrite() in the STM run-time.

We use a polite contention manager [58], and in order to avoid infinitely running transactions due to inconsistent reads we validate the read set by inserting a call to validateReadSet on back edges and method entries. This performs validation once every 1,000 calls. Arrays are synchronized on TxnRecords held by a global array, since we cannot force them to extend a super-class of our choosing. Our runtime system uses the array's hash code in order to index into the global array. This will occasionally cause accesses of disjoint arrays to be perceived as contention.

Finally, and in order to make our evaluation more realistic, our implementation performs some basic optimizations on both the base case and the optimized case. We do not open the receiver object for reading on an access to a final field. Also, we perform a basic intra-procedural flow analysis to remove redundant read and write open operations on the same object.

### 8.2.2 Optimization

In this section we describe a technique for statically optimizing the performance of programs annotated with access permissions. In this section we describe the optimization process, while in Section 8.2.3 we discuss some of the implications of this process.

Our optimization occurs during source to source translation, as in-transaction reads and writes are encountered. We use the access permission associated with the object reference to determine if we *really* have to open the object for reading or writing. The reason that we open an object for reading or writing is to protect it against concurrent access by multiple threads. So if an object is not thread-shared, then it does not need to be opened for reading or writing. Fortunately for us, in Chapters 3 and 4, we showed that access permissions can soundly approximate whether or not a given reference points to a thread-shared object.

Recall that objects referenced with share permission are assumed to be thread-shared, while objects referenced with unique permission can be treated as thread-local. In the latter case, our optimization will not open the object for reading or writing. The first three rules of our optimization (naively) assume that the access permission alone is a sound approximation of thread-sharing:

**Rule 1** References of immutable permission will never be opened for reading. Since no thread will change their value, there is no need to protect a thread from concurrent modification.

**Rule 2** When writing to the fields of a unique object, it is not necessary to open that object for writing since no other thread can concurrently access the object. However, it is necessary to log the initial value of the object as the transaction may still be rolled back. Therefore, when writing to objects of this permission, a call to the `txnOnlyLogWriteObject` method is inserted, which logs a copy of the object, but does not perform an atomic test-and-set on its owner field.

**Rule 3** Neither objects of unique nor full permission ever need to be opened for reading. Again, since no other threads have modifying permission to objects of these permission kinds, there is no need to protect our thread from concurrent modification.

Of course an object that is uniquely referenced by the field of a thread-shared object becomes thread-shared itself! If our optimization just consisted of these first three rules, these thread-shared objects would not be protected from concurrent modification.

Therefore, we add one additional rule to ensure that our optimization is sound. The net result will be that either a unique object was *actually* thread-local, or that the uniquely referenced object has become part of the synchronization domain of another thread-shared object.

**Rule 4** Because unique and full permissions can be reached through fields of other thread-shared objects, we require that any share, full, or pure object be opened for writing before any method is called on a unique or full field of that object.

The first and third rules will lead to a reduction in the number of synchronizing operations in the resulting translated program, since no check will be performed to query the "owned" status of that object. These rules will also lead to a reduction in the number of logging events, since their consistency will not need to be later checked. While logging is a thread-local operation, it does require inserting an item into a hash table. The second rule will help to eliminate the

synchronization overhead of an atomic test-and-set, which is required when acquiring ownership of an object.

Note that references associated with full permission still must be opened for writing, as other pure references may be used to concurrently read the same object.

In Figure 8.3 we have illustrated the effect of our optimization on the `contains` method of a linked list. This linked list is used for the buckets of a hash set, which we use as a benchmark and describe in detail in Section 8.3. Since the list is singly-linked, each element of the list holds a unique permission to the next element. When the `contains` method is called, each element first checks to see if it contains the requested element before calling the same method on the next element in the list. As the `@Imm` annotation indicates, an immutable permission is required to call the `contains` method. This requirement can be satisfied by the unique permission that each element holds to the next element in the list.

The primary difference between the optimized and unoptimized versions of this method is the removal of the call to `__aj_get_value(...)` in the optimized version. This call would normally open `this` for reading, but since we have a unique permission to the list node, we do not require synchronization. Also, note that subsequent reads on fields of the receiver do not perform synchronization in either case, because of our basic optimizations.

In the next section we further discuss the ramifications of our changes.

```
@Imm boolean contains(@Pure Object item) {
  if( this.value.equals(item) )
    return true;
  else if( next == null ) return false;
  else return next.contains(item);
}
```

```
boolean contains_atomic(Object item) throws TransactionException {
  txnPeriodicValidation();
  if (UniqueLinkedList.__aj_get_value(this).equals_atomic(item))
    return true;
  else if (next == null) return false;
  else return next.contains_atomic(item);
}
```

```
boolean contains_atomic(Object item) throws TransactionException {
  txnPeriodicValidation();
  if (this.value.equals_atomic(item))
    return true;
  else if (next == null) return false;
  else return next.contains_atomic(item);
}
```

Figure 8.3: The `contains` method of a linked list, before translation (top), and as translated for use in atomic contexts without (middle) and with (bottom) optimization.

### 8.2.3 Discussion

We have presented a technique for optimizing the performance of STM programs using access permissions that will potentially reduce overhead on thread-local, immutable objects, and other objects that are used in restricted aliasing patterns. However, there are some more subtle points that deserve further discussion.

The first thing to note is that while we can reduce or even eliminate the overhead associated with reading and writing references of immutable or unique permission, those are the sorts of operations that, by themselves, do not need to be performed inside of an atomic block at all. And, of course, our static analysis (NIMBY, an analysis related to Sync-or-Swim but for atomic blocks) tells us statically where atomic blocks are and are not necessary. So we mainly expect to see performance improvements for unique and immutable objects that are incidentally accessed inside of atomic blocks due to actions being performed on other, thread-shared, objects.

Next, it is interesting to elaborate on the point made in the previous section: objects referenced through unique reference are not necessarily thread-local. Rule 4 ensures that even if a uniquely referenced object is thread-shared, it will still be protected from concurrent access, namely because all threads that will have accessed it will already have had to open the referring object for writing, which can only be performed by one thread at a time. But because many objects can be protected through ownership of one outer object, there is the potential to greatly reduce overhead in some programs.

Occasionally, because of Rule 4, our optimization may insert "open for write" operations that were not otherwise necessary. Therefore, we must ask if the potential increase in contention is worth the reduction in overhead. Recent work has suggested that overhead, not contention, is the primary cause of poor performance in STM implementations [33]. For programs that generally access disjoint regions of memory, the increased granularity will hopefully not matter. We specified our HashSet benchmark (Section 8.3) twice in order to observe the effect of this increased granularity, and saw that, as expected, overhead was lowered but contention increased as the number of threads increased. Interestingly, the performance is not dramatically worse even in such a program with artificially high contention.

Finally, it is interesting to point out that sometimes with our system, a programmer's specification goals may conflict with his performance goals. When writing a method specification for the purposes of behavioral verification, a programmer generally wants to write the weakest precondition possible. This will make the method useful in the largest number of contexts. In our system, this means using a pure or share permission. However, when performing optimization, since we must assume conservatively that references of pure or share permissions are thread-shared, this may result in under-performance when a stronger permission was available. For example, if the programmer has unique permission to an object, they would like pure method calls on that object to not require any synchronization. This is a natural use for method specialization, since, statically, we can identify the points at which a caller has a stronger permission than strictly required by the method. Creating a copy of that method with reduced synchronization would help improve program performance. While we have not implemented this specialization feature in AtomicPower, we plan to do so in the future. Some of our benchmarks have been specialized by hand in order to take advantage of this observation. We make these cases explicit in the next section.

## 8.3 Evaluation

In order to evaluate our technique, the optimization was performed on a suite of annotated benchmarks of varying sizes. Performance of these optimized programs was compared to performance in our baseline implementation. In this section we describe the results of these benchmarks. We also describe our experiences specifying these concurrent programs and report on interesting patterns.

### 8.3.1 Methodology

For the purposes of evaluation, we chose several benchmarks. Our suite consists of micro-benchmarks, popular STM benchmarks, and an open-source Java video game. For programs that were not originally written to use atomic blocks, we replaced existing synchronization constructs. When doing this, we attempted to the best of our ability to mimic the synchronization style of the original program.

Next, we used access permissions to specify as many of the methods and classes as possible, in order to describe the program's aliasing behavior. This required a good understanding of each program's run-time behavior. After specification, we used NIMBY [10], our static permission checker, to check the consistency of our specifications. NIMBY is a version of the Sync-or-Swim static checker that is aware of the semantics of atomic blocks. The verification process ensures that the access permissions specifications were actually correct. While the primary goal of NIMBY is to check object protocol use, we did not specify any for the purposes of this experiment. Table 8.1 describes the number and type of full, unique and immutable permissions that were used in each benchmark, since these permissions are the ones that provide performance benefit. For the largest benchmarks, we did not specify all of the references in the system. Specifically, we ignored methods and objects that were never used in transactions and we did not specify methods of pure or share permission that did not interact with other permissions in meaningful ways.

| Benchmark | Refs. Annotated | | | Open Calls Removed (Total) | | Extra OW |
|---|---|---|---|---|---|---|
| | immutable | unique | full | read | write | Calls Inserted |
| ReadHeavy | 2 | 2 | 0 | 1 (1) | 0 (0) | 0 |
| WriteHeavy | 0 | 4 | 0 | 0 (0) | 1 (1) | 0 |
| ListSet | 0 | 5 | 0 | 4 (19) | 2 (18) | 0 |
| HashSet | 0 | 4 | 0 | 1 (16) | 0 (5) | 1 |
| 4InALine | 124 | 23 | 1 | 41 (289) | 8 (100) | 1 |

Table 8.1: Number of references annotated with helpful access permissions, and the number of open for read/write calls this removed. The last column lists the number of additional open for write calls inserted due to rule 4.

After permission verification, we took each benchmark and ran it through AtomicPower, our source-to-source translator, with and without our permission optimizations. For each benchmark, our optimization removed a different number of calls into the STM run-time system. Table 8.1

describes the number of open for read and write calls that were statically removed for each benchmark, as well as the number of additional open for write calls that were inserted. Note that in general the removal and insertion of STM operations at different locations in the source program will have a different effect on overall benchmark performance.

In general our STM implementation is not sound unless it is used to translate every file in an application. However, some of our benchmarks used classes from the Java standard library. While many of these classes could be translated from source, some could not due to limitations in our source to source translation (primarily due to use of anonymous inner classes and some features of Java generics). In a few cases we created new implementations which could more readily be translated by AtomicPower. In such cases we attempted to be as faithful as possible to the original implementation.

Each benchmark has its own measure of performance, usually elapsed time or number of operations performed. We ran each with and without optimizations for 1,000 runs (unless otherwise noted), varying the number of threads as we went along. All of our performance numbers come from executing programs on a Dell PowerEdge 2900 III with 2 Quad Core Intel Xeon X5460 processors, running at 3.16GHz (1333MHz FSB) with 2x6MB of L1 cache, 32 GB of RAM, and running Linux 2.6.23.1-001-PSC and Sun's Java SE Run-time Environment (build 1.6.0_07-b06).

We will now briefly describe each benchmark in turn.[3]

**ReadHeavyTest and WriteHeavyTest** In order to get a feel for the potential of our optimization, we created two synthetic benchmarks, ReadHeavyTest and WriteHeavyTest. Both programs access objects inside of a transaction, but do so with only a single thread. ReadHeavyTest creates a chain of objects, each of which refers to the next with immutable permission, and then inside of a transaction reads from fields of every object in the chain. The entire process is performed 1,000 times inside of a loop, and was designed to illustrate the effect of removing an open for read operation. WriteHeavyTest is the same, except that each object in the chain refers to the next object with a unique permission, and during the transaction each object in the chain is modified. This benchmark was designed to give us a feel for the amount of overhead that can be reduced when removing the ownership acquire operation, but retaining the object copy operation. For comparison purposes we also ran the same two experiments without any synchronization.

**ListSet** ListSet was used as an STM benchmark in a paper by Herlihy et al. [59]. The data structure itself is a set, implemented as a doubly-linked list. This list was note-worthy in two ways. First, because it is doubly-linked, each element of the list has share permission to both its successor and predecessor. This makes sense, since (almost) every element of the list is referenced twice. If, instead, a singly-linked list were being used, we might be able to gain some performance benefit from their unique permissions. Second, this benchmark is interesting because it creates local objects inside of transactions that escape from their allocation context and are later accessed, but are not shared with other threads. At the beginning of the benchmark a number of threads are spawned. Over the course of two seconds, each thread attempts to randomly insert, remove and check membership of random items. We measured the number of

[3] All of the benchmarks are available at `http://www.nelsbeckman.com/research/atomicpower/`.

operations accomplished by each thread. Each thread performed 30% updating operations and 70% reading operations.

**HashSet** We created our own implementation of a hash set for benchmarking purposes. In this implementation, the hash set holds an array of bucket nodes that each point to a linked list. Inside the linked list, each node points to the next with unique permission. The outer object, however, points to each bucket node with share permission so that it will not become a contention bottleneck. In order to evaluate the effects of Rule 4, which may occasionally insert extra open for write operations, we also specified a "high contention" version of the same program. In this version, the outer hash set object points to its buckets with unique permission. This will eliminate synchronization internal to the data structure, but will effectively serialize access to it, since the outermost object will alternatively owned by each transaction, preventing all other transactions from accessing the set. Both versions pass the permission checker, so both specifications are sound. For this benchmark, we created a number of threads and made each perform 100,000 operations, 30% of which were updating. We measured the elapsed time.

**4InALine** We wanted to evaluate our optimizations on a real program representative of common multi-threaded OO programs. For this purpose, we chose 4InALine[4], a GUI-based video game that is a clone of the board game Connect Four. We chose this program because it was relatively large (5471 LOC in 62 classes), it was well-designed and documented, and seemed at first glance to contain a number of immutable and thread-local objects that were being accessed inside of critical regions. 4InALine stores shared game data in a server object that is accessed by client threads, one per each player in the game, and by a GUI update thread. These threads will each occasionally make a copy of the current game board, which they use to either calculate a next move, or to determine the visual representation of the board.

4InALine required some modification before it could be used as a benchmark. We replaced synchronized blocks and wait/notify statements with atomic blocks (57) and a retry statement (1). This program uses JFrame, a Swing framework class which allows users to create GUI windows. We created a wrapper class that would be introduced as an intermediary by AtomicPower. This wrapper ensures that user subclasses of JFrame will be properly synchronized without requiring us to translate large portions of the Swing framework. In practice, this translation strategy worked well, resulting in a program without flickering or obvious synchronization defects.

For the experiment, we ran 4InALine in a deterministic AI versus AI game on the weakest difficulty level, and gathered the elapsed time from game start to completion.

### 8.3.2 Results and Discussion

The results of our benchmarks are shown in Figures 8.4 through 8.7. In general, our optimizations improved performance, although to varying degrees. Most improvements can be attributed to unique and immutable references.

The results from the ReadHeavyTest and the WriteHeavyTest (Figure 8.4), show that there is potentially a great deal to be gained by optimizing access to unique and immutable objects.

[4]http://code.google.com/p/fourinaline/

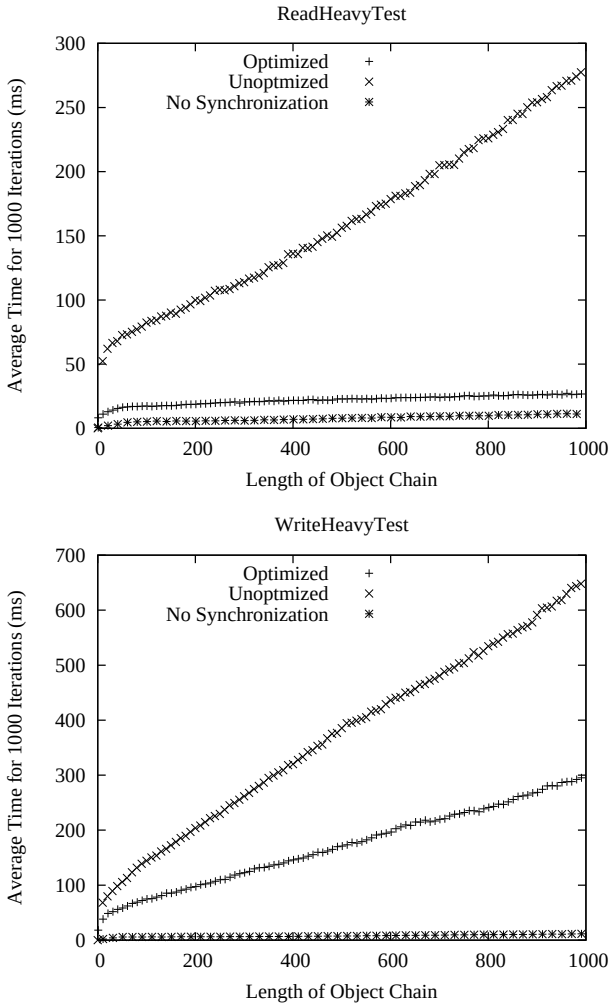


Figure 8.4: The results from running ReadHeavyTest and WriteHeavyTest (less is better).

In particular, removing the open for read operation provides a big benefit, since this makes a memory read essentially free. The synchronization-free benchmark is always faster even for the read-only case, since there is some overhead associated with starting and committing the 1,000 transactions that are performed during each run.

The performance of ListSet (Figure 8.5) was improved because it uses a number of thread-local objects that happen to be accessed inside of atomic blocks. ListSet creates a Neighborhood object on each look-up. This object escapes its allocation context, but is immediately used by the caller, which is still inside a transaction, to determine the result of a search. This process happens once per operation.

However, in our system, objects do not have to be thread-local to be optimized. Uniquely referred objects can still be part of a thread-shared data structure, such as the bucket lists in the HashSet benchmark (Figure 8.6). Because the randomized inserts, contains and remove operations generally hash to different buckets, threads do not contend, and therefore the overhead that is saved because the entire linked list is being locked once at the head results in better perfor-

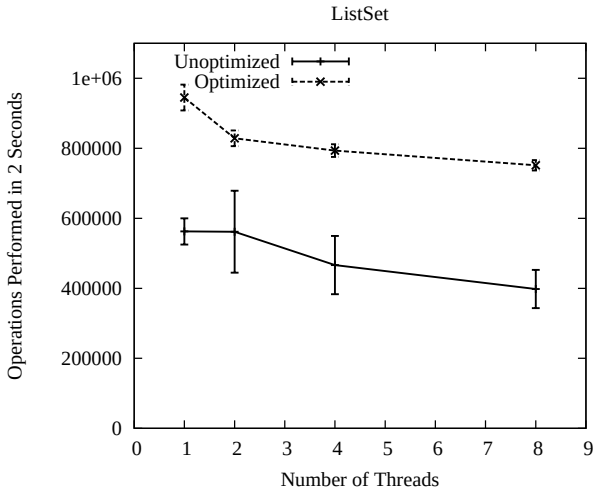


Figure 8.5: Mean number of total operations performed in 2 seconds in the ListSet benchmark, using 30% modifying operations (more is better).

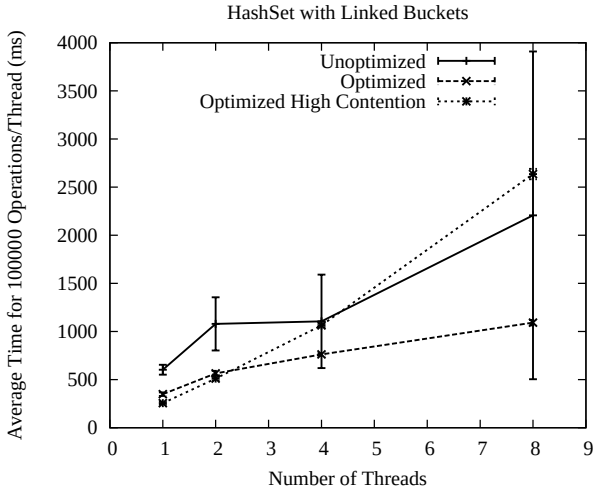


Figure 8.6: Mean completion times for the HashSet benchmark for different numbers of threads, using 30% modifying operations (less is better). Note the large standard deviation for the unoptimized case.

mance. Furthermore, note the large standard deviation for the unoptimized case. We speculate that this is due to transaction aborts, which are generally expensive. Because the buckets are locked at the front, aborts are extremely rare in the optimized case, but can occur in the unoptimized case, where a thread may traverse the list, have it modified behind it, and then be forced to abort since its read set is now out of date. For our high contention specification, as expected, overall performance is better for smaller numbers of threads, since almost all synchronization operations will be removed, but degrades as more threads attempt to access the data structure and the single lock becomes a bottleneck.

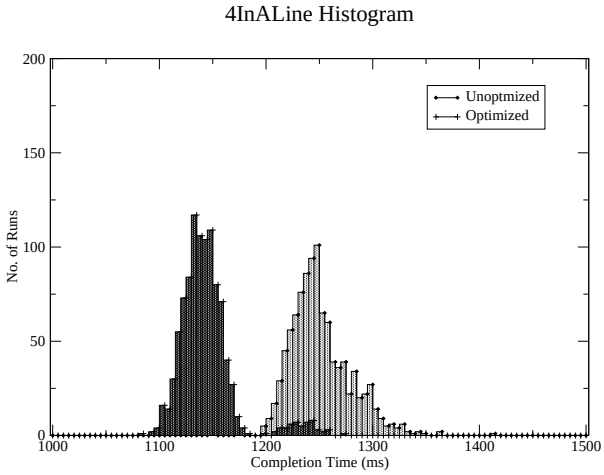


Figure 8.7: Histogram of completion times for 4InALine (left is better, x axis begins at 1,000).

4InALine (Figure 8.7) benefits from its use of a number of immutable objects. There are many pieces inside the model (which itself is thread-shared and mutated) that are never modified, and therefore numerous reading methods, such as calls to equals, are sped up. Also, 4InALine uses a number of immutable collections, such as a cache for storing lines that are known to be winning lines. Each line is implemented as an immutable list of immutable pieces, although to take full advantage of immutability, we had to perform hand-specialization, copying certain methods and re-specifying them as taking an immutable receiver.

## 8.4 Related Work

There has been much previous research attempting to optimize the performance of software transactional memory and to reduce its overhead.

For instance, work has been done in statically identifying objects that were allocated inside of a transaction using a whole program analysis [1, 55]. Shpeisman et al. [87] use a whole-program alias analysis in order to identify objects that are never accessed inside of a transaction, and additionally perform a dynamic escape analysis in order to find thread-local objects. Aldrich et al. [3], Blanchet [19] and Choi et al. [27] also perform an inter-procedural analysis in order to identify synchronization operations that can be removed, although not in a TM context.

Our work is different in a few ways. First, all of our optimizations are performed statically. Most importantly, our approach is modular, and uses only intra-procedural analysis. This is feasible because of the static access permissions which are provided by programmers, and checked for correctness. This may make it easier for our approach to scale to very large applications. Moreover, our approach is consistent with a language that uses dynamically linked libraries. As long as the code that we link against has been annotated, or we can do so externally, the optimizations we perform on our own code will be sound. Our analysis is sometimes more precise than existing approaches, because the designer's intent is encoded in the annotations. For example, objects stored exclusively as fields of Thread objects can indeed be treated as thread-local. In

earlier work, Shpeisman et al. [87] noted that fields of a thread could not necessarily be optimized as thread-local, since a new thread object is always reachable from its spawning context. This reduced their opportunities for optimization. In our approach, the `start` method on a thread can be specified as consuming the entire unique permission to the thread object. Figure 8.8 shows just such an example. This prevents the spawning thread from modifying or reading the newly created thread, thus providing us with another opportunity for optimization.

```
class ConsumerThread {
  @Unique(returned=false)
  void start() { super.start(); }

  @Unique void run() {
    atomic: {
      Object i = this.input.get();
      doWork(i);
      this.output.put(i);
    }
  }
}

void spawnConsumer() {
  ConsumerThread t =
    new ConsumerThread();
  t.start();
  //Cannot access thread object
}
```

Figure 8.8: The `start` method of the ConsumerThread class consumes the entire unique permission produced at construction-time. As a result, ConsumerThread need not be opened when reading and writing its fields inside an atomic block.

## 8.5 Conclusion

In this chapter we presented a static technique for reducing the overhead of software transactional memory based on our system of access permission. The information provided statically by access permissions allows us to remove unnecessary synchronization and logging operations that traditionally require a whole-program analysis. Moreover, because access permissions are also useful for verification, programmers willing to use our behavioral specifications can take advantage of our optimizations without additional effort. We have implemented our technique in a tool called AtomicPower, and showed improved performance on a number of benchmarks.

# Chapter 9

# Conclusion

*You're at that critical point. Make that last push.*

---

## 9.1 Hypotheses and Thesis Statement, Revisited

This thesis presented a static analysis capable of checking that object protocols are used correctly in concurrent programs. That static analysis was the embodiment of a larger methodology at the core of which are access permissions, flow-sensitive type qualifiers that track object states, and succinctly describe the ways in which the referenced object may be aliased. Due to gaps in expressiveness, we extended the original access permissions methodology with bounded parametric polymorphism. Such a feature allows succinct specification of classes and methods that are ambivalent to the sorts of permissions with which they interact, and is needed frequently to specify collections and reusable runtime structures.

The overall approach was motivated by an empirical study that showed a high frequency of object protocols in practice, including a number of types designed to be used in multithreaded applications. Because the analysis is modular, it depends heavily upon specification. To mitigate the potential for unreasonable programmer burden, we introduced Anek, a specification inference tool based on probabilistic constraints. Those constraints encode which specifications are likely to be good in a particular scenario and can help to make specification inference more robust in the face of bugs.

Access permissions were even used to optimize the run-time performance of a software transactional memory system, showing a certain amount of symbiosis between program verification and performance.

Now that the main body of the thesis has been presented, let us revisit the hypotheses, and the main thesis statement, to see how they have been validated.

### 9.1.1 Hypotheses

#### Hypothesis 1: Prevalence of Object Protocols

Object protocols are an important and recurrent pattern in object-oriented development, and therefore are worthy of further study.

**Validation** In order to validate this hypothesis I completed a large study of open-source Java software in order to determine the nature and frequency of object protocols as they occur, "in the wild."

**Result** This hypothesis was upheld. During the course of the empirical study, it was found that object protocol definition is rather common, occurring in about 7% of all types in the corpus. Use of protocols is even more common, occurring in about 13% of all types. So given that protocols are relevant to roughly one in ten classes, this means that the developer of an average application will come into contact with them many times. Of the protocols that we found, 25% were found inside of classes designed to be accessed by multiple threads, meaning that correct protocol use in concurrent programs is also important. Finally, seven behavioral categories of object protocols were discovered which covered 98% of the protocols encountered. This means that there is a great deal of similarity in the types of protocols that are occurring in the wild.

#### Hypothesis 2: Formalization

We can develop and formalize an analysis that will guarantee a concurrent program does not violate the object protocols that it defines and prove that the system will not produce false negatives.

**Validation** This hypothesis was validated by developing and formalizing a type system and operational semantics based on our permission system and proving the type system sound with respect to its semantics. The proof essentially says that no object in a program will ever be required to be in some abstract state that at run-time it will not actually be in.

**Result** This hypothesis was also upheld. In Chapters 3 and 5 a language and an associated type system were formalized which allowed the definition and checking of object protocols and their use. In Chapter 4 a proof of type safety for a core of this language with respect to its operational semantics was presented. This proof of soundness guarantees that the protocols that are defined in the language will not be violated at run-time as long as the program in question passes the typechecker. Certain aspects of the system, for example protocol dimensions and parametric polymorphism, were not included in the proof of soundness. Nonetheless, it is our claim that the proof itself includes the essential aspects of the language, and that the proof of soundness for the parts omitted would be largely straightforward and/or is similar to existing work.

### Hypothesis 3: Specification Coverage

Our specification system can be used to specify the behavior and implementation of object protocols in real concurrent, object-oriented programs.

**Validation** In order to validate this hypothesis, I have specified the behavior of object protocols in numerous small and two large concurrent Java programs collected from open-source projects. During this process I noted and here report the recurring and interesting patterns of protocols that can and cannot be specified.

**Result** This hypothesis was upheld. The case study itself was described in Chapter 6. In this chapter the expressiveness of the type system is discussed. Thanks to state hierarchies and state dimensions, the specification language did an excellent job of describing the protocols to be checked. In all cases it was able to accurately describe the protocol itself. Regarding state invariants, Sync-or-Swim also performed well, and with the exception of a technical issue having to do with locations at which Java annotations are permitted, all state invariants were able to be expressed. The permission kinds themselves were also expressive enough to cover most of the thread sharing and aliasing patterns encountered. While it could be argued that some of the imprecisions we encountered could have been remedied with more expressive permissions, it is more likely that the imprecisions are due to a combination of overly restrictive rules and bugs in our current implementation. The inability of programmers to specify which lock protected which piece of memory was never an issue, although this likely has to do with our selection of case study programs. We expect that other programs might reveal different results.

### Hypothesis 4: Analysis Precision

Our analysis will report a relatively low number of false positives, on the order of the number of false-positives reported by comparable automated behavioral analyses.

**Validation** In order to validate this hypothesis, I have built an automated static analysis for Java and used it to check the specifications on the suite of case studies. The rate of false positives per line of source was noted and compared with other approaches.

**Result** This hypothesis was largely upheld. The rate of false positives, that is falsely reported protocol violations, during our case studies varied from 1 per 1,259 lines of source to 1 per 2,238 lines of source. This is comparable to that reported by similar approaches. Still, we saw more false positives than we would have liked. The false positives themselves, not including unnecessary mutual exclusion, were largely due to bugs in our implementation, features that are currently unsupported but easily added, and iterator-like protocols where the non-emptiness of some collection is known but not expressed within our analysis.

**Hypothesis 5: Mutual Exclusion Requirements**

In order for a program to be verified, it should not require a great deal more critical sections than is strictly necessary for functional correctness.

**Validation** During the verification process I have observed and reported on the number of times that my analysis forced me to add synchronization in the cases where the original programs were synchronized correctly.

**Result** This hypothesis was also largely upheld. In general, it was very rare for Sync-or-Swim to force the addition of mutual exclusion. The primary exception is a reusable tree data structure whose protocol was checked during the JabRef case study. This tree had a protocol that was uninteresting with respect to the JabRef program. This made specification easy, since we were able to use a guaranteed share permission. But due to the strong synchronization requirements of our system, this specification technique resulted in our being forced to add synchronization in numerous locations. This is an area that could likely be addressed fruitfully in future work.

**Hypothesis 6: Probabilistic Inference**

Probabilistic specification inference, an inference which can encode intuitions describing common or "good" specifications, is a good solution for reducing programmer annotation burden, resulting in specifications with comparable false-positive rates as those written by hand.

**Validation** In order to evaluate this hypothesis we have created a probabilistic inference tool Anek, and evaluated its performance inferring specifications for a large open-source program. The specifications inferred were compared with ones written by hand for both subjective quality and the number of false positives they gave rise to when subsequently running our protocol checker.

**Result** This hypothesis was upheld. In our case study Anek performed well. The specifications it inferred generated just one additional false positive when compared to specifications written by hand. Anek showed itself to be robust to inconsistent constraints by generating good specifications (similar to those written by hand) even when the underlying technique was not precise enough to verify a protocol use. Moreover, the specifications generated were found to be well-written, closely mimicking what experienced developers might have written by hand.

**Hypothesis 7: Optimization**

Because access permissions describe aliasing behaviors, permission annotations can be used to optimize transactional memory, improving its performance.

**Validation** In order to validate this hypothesis, I modified a source-to-source implementation of transactional memory for Java to remove unnecessary synchronization and logging based on the access permission annotations. I compared performance of this optimization by running a benchmark suite with and without the change.

**Result** This hypothesis was largely upheld. When comparing the optimized benchmarks to the original benchmarks performance was generally improved, ranging from a 10% improvement to a 40% improvement. Still, results suggest that in certain types of applications, those with high thread contention, performance may not be as good. This is an area left for future investigation.

### 9.1.2 Thesis Statement

The validation of these seven hypotheses supports the original thesis statement:

> Access permissions, which statically describe the aliasing behavior of program references in object-oriented programs, provide a good basis for the lightweight verification of object protocols in concurrent systems, allowing us to verify real programs and optimize the underlying run-time system.

After showing that protocols were prevalent in practice, we developed a static analysis for checking their correct concurrent use. In our case studies, which used this analysis to check open-source programs, we found that this approach was quite good, allowing for easy expression of the relevant protocols, a low rate of false positives without significant programmer burden. To further lower the programmer burden we presented a specification analysis tool, and then we showed of the potential performance improvements available with access permissions by optimizing an STM run-time system. All in all, the investigation of access permission for the purposes of concurrent protocol checking was quite fruitful.

## 9.2 Challenges

Of course, not everything is perfect, and the application of access permissions to concurrent protocol checking came with its own share of headaches. Let us now reiterate some of the most troublesome issues encountered.

**Confirming Protocol Definitions** As it turned out, confirming protocol definitions was not a very productive use of time. In Chapter 2 we presented the methodology of our empirical study, and we described a simple static analysis, the ProtocolFinder, used to find evidence of object protocols. Unfortunately, this static analysis was not extremely precise, so as part of our study we manually inspected the results of the ProtocolFinder to classify each warning as either a true positive, actual evidence of an object protocol, or a false positive. This process ended up being very time-consuming, and in the end potentially led the study to be less comprehensive than is ideal.

The job of investigating these warnings fell to Duri Kim, the masters student spearheading the project. And it took roughly a month to investigate the nearly 2,000 warnings returned by the

ProtocolFinder when it was run on our four case study programs. The amount of code examined was large, roughly one million lines. Still, an incredible effort was spent and this effort can largely be chalked up to the imprecision of our ProtocolFinder.

This is not simply an issue of avoiding work. Because of the amount of time spent investigating the warnings returned for the first four programs, we were not able to add any additional programs to the protocol definition phase of the empirical study. We would have liked to investigate many more programs to get a better handle on the frequency and nature of the object protocols they defined. Doing more programs would also allow us to see larger trends, like the rate of protocol definition in libraries versus frameworks versus free-standing applications. A more precise protocol detector would have likely enabled us to investigate more programs in the same amount of time.

One reasonable option would have been to use the results of existing research, taking someone else's protocol finder rather than developing our own. Recent work has pushed the boundaries of automatic protocol detection [60, 96, 101], and by using an existing tool with a good false-positive rate, we could have both exploited the full body of research and spent more time examining the characteristics of these protocols. Still, all of these tools are built with a specific definition of object protocol in mind that may be slightly different from our own and may therefore miss some protocols.

**Restrictive Type-Checking Rules** When it came to Sync-or-Swim false-positives there were certain repeating themes. Three broad classes of issues led us both to more work and more imprecision than ideal. Those classes are field specifications, unnecessary synchronization, and the need to prepare for object reentrancy.

As mentioned in Chapter 6, the need to specify fields with access permissions simply makes life harder. This is not a problem with the concept state invariants per se, or even a lack of expressiveness in invariant specifications. Rather, it is the simple fact that when protocol classes are ubiquitously referred to by fields, one ends up doing a lot more work. A programmer starts by adding a state invariant to one class because that class calls a permission-requiring method on a field. This in turn requires that the method in which the field is accessed be specified to require a permission. This then requires finding all the callers of the method to make sure enough permission is available at the calling context. And if that method is called on a field, the process can repeat again and again. There is no immediate good solution for this other than possibly having better default permissions. However, one sign that is at least slightly encouraging is that it appears protocol-defining types are less likely to be used as fields. This is possibly due to the difficulty in ensuring that a protocol is obeyed for an object that is assigned into the heap in such a manner.

We also discussed unnecessary synchronization in Chapter 6. Sadly, the majority of the unnecessary synchronization came in situations where the protocol under verification was rather uninteresting. In the JabRef case study, this was in the verification of the mutable tree API. In all uses of the mutable tree, the instances themselves were never set to 'immutable mode.' Instances were freely mutated, and while some instances were shared amongst threads, many were not. For this reason we specified a a guaranteed share permission every time the mutable tree was used. But because our rules require us to synchronize the method receiver whenever a share object is

unpacked, this resulted in a lot of unnecessary synchronization. This was particularly frustrating because of a common pattern: share objects were often unpacked for the sole purpose of calling a method on a share field of that object. But as soon as that object is accessed it will have to synchronize as well, rendering the outer synchronization unnecessary. Through this repeating pattern, we realized that synchronization is only necessary when unpacking objects if one intends to access unique or full fields. In other cases the rules are overly restrictive.

Finally, during the several years that I have verified programs with access permissions, the need to prepare objects for reentrancy has been an continual source of difficulty. Before each method call site, any non-unique unpacked objects in the calling context must be packed. This is to prepare those objects for the possibility that the method being called will transitively call back into the unpacked objects. Access permissions always provide a consistent view of the object, which can be contrasted with other approaches like Universe Types [35]. This is good because it allows the specification of certain common patterns, like dynamic state test methods. But many times no invariant of an object can be satisfied before a method call, precisely because the original programmer knew reentrant calls were impossible. Our system still requires packing even in such cases. To get around this problem, Sync-or-Swim does provide an unchecked `@NonReentrant` annotation, but a far better answer would be a sound solution that allowed some objects to be reentrant while prohibiting others.

**Mixed Optimization Performance** While our permission-based optimizations did generally improve performance, the HashSet benchmark leaves us to believe that in certain situations the optimized implementation may actually perform worse than the unoptimized version. Ideally we would like our optimization to at least be on par with the original version when it is not better. But occasionally our optimization will lead to increased thread contention. The logic behind the optimization is one of overhead savings. If unique objects can be treated as thread-local objects, then the bookkeeping that is typically required for each object in run-time (e.g., logging, copying and locking) can be removed. In order to enable this, some share objects may end up being locked when the would not be required to in the unoptimized version. This can improve efficiency overall by using single locks to protect large groups of objects. But it may also lead to locking bottlenecks, particularly in high contention programs. The HashSet benchmark, discussed in Chapter 8, is exemplary of this undesirable behavior.

There is good news. Prevailing wisdom says that thread contention is low in most multi-threaded programs. It took an enormous amount of thread contention in the HashSet benchmark to exhibit this slowdown (each thread essentially did no work other than accessing the shared data structure). Nonetheless, some programs do exhibit high contention and this may be a weakness in our approach.

## 9.3 Future Work

With every challenge comes the opportunity to make improvements and to advance the state of the art. Here we will discuss some of the most intriguing possibilities for future work, both long-term and more immediate.

### 9.3.1 Immediate Improvements

Many of the challenges we encountered suggest future opportunities. For example, the transactional memory optimizations embodied by the AtomicPower tool could be improved for high contention scenarios. Specifically, the optimization could be disabled for those objects that show high thread contention. STM runtimes already contain numerous data structures for monitoring the behavior of threads and recording the memory that they access. Perhaps with a simple extension one could track objects that are highly contended. For these objects, dynamically disabling our optimization might prevent threads from acquiring locks that they do not really require, which can cause bottlenecks. Such a dynamic solution would still allow programmers to gain the benefits of optimization for objects that are not highly contended.

The empirical study presented here represents an important step in our understanding of object protocols in the wild, but more can be done. To gain more confidence in our understanding of the types and frequency of object protocol definitions, it would be good to expand the scope of the study, bringing in a larger number and variety of candidate programs. The next step after understanding these aspects of protocols is understanding their weaknesses. Do programmers have trouble obeying protocols in practice? Does the widespread use of object protocols make a program more buggy? Some ways of exploring these questions are with more inventive corpus studies. For example, one could attempt to correlate bug logs with rate of protocol usage or or observe programmers' difficulties as they attempt to use protocol-rich APIs.

It would be interesting to expand the scope of specification inference beyond what is currently possible in Anek. There are significant opportunities for decreasing programmer burden in this area. By design, Anek's ability to infer state invariants is limited. Unfortunately, state invariants in practice are the most difficult parts of program specification. Anek's limitations are largely due to its scope. Anek was never meant to infer protocol states. Part of the difficulty of writing a state invariant is actually deciding how many states an object can have, and how the state invariants of each of those states can work together to provide exactly the field permissions that are necessary at any given time. One natural approach for improving Anek, then, would be to combine it with an existing protocol inference tool [60, 101]. The protocol inference tool could take on the responsibility of deciding which abstract states an object has, while Anek could have the responsibility of determining the invariants to be associated with each state.

A few suggestions have been mentioned in this thesis that might ultimately improve the precision of the static analysis itself. The language could be improved in an sound way so that non-reentrant objects would not be subject to the same packing constraints as possibly reentrant ones. The unpacking rules might be changed to reduce the amount of unnecessary synchronization. One might even be able to tease apart separate permissions so that thread-shared permissions and aliased, thread-local permissions could be distinguished. All of these improvements would make the analysis more pleasant to use, although it is not immediately obvious how to enact them.

Finally, the soundness of certain parts of the formal system were left unproven, notably state hierarchies and polymorphic permissions. We believe that these features are sound in combination with the proven features and a full proof of type safety would have provided little return on the investment, given either the nature of the feature or the seeming similarity of a proof of soundness to that in existing work. Still, a full proof of type safety for the complete language

presented in this thesis would increase confidence in the overall approach.

### 9.3.2 The Long Term

In the long term, follow-on work of a more lasting nature can be imagined. The experiences of this author give us a number of issues to contemplate about the present and future state of programming practice. The first is protocols themselves. While typestate checkers and object protocols are familiar to those steeped in academic literature, they are much more foreign to students and practicing programmers. While most programmers have likely used an API that defines a protocol, it is unlikely that they gave that particular experience any significant thought. This is undoubtedly because object protocols are not presented to students and in the mainstream programming press as well-defined notions. But in fact, the act of using an API that defines a protocol is a somewhat unusual experience. Clients must now be aware of two additional pieces of information related to the use of the API. What abstract states are defined by this API, and which methods of the API place requirements on the abstract states of the objects they accept? It is my hope that in the future students of computer science will leave the university with a better conception of object protocols so that their encounters with protocols will be less surprising and more principled. To that end, this thesis fills in more gaps in our understanding of object protocols.

Better understanding of protocols may have another benefit: fewer unnecessary object protocols. It is this author's belief that certain APIs define protocols that are absolutely necessary. Often this is the case when an API is a wrapper for some underlying resource that is inherently stateful, for example a socket or an abstraction of a piece of hardware. However, it also seems to be the case that some APIs define unnecessary protocols. Their authors may create these protocols through ignorance or even lack of care. Given the trouble that is required to successfully reason about protocol use, it seems that the prudent thing to do would be to avoid defining them whenever possible. As more and more programmers come to know about protocols and recognize their difficulties, I would expect fewer APIs that are unnecessarily difficult to use. Of course, this all must be taken with a grain of salt. Recent work [91, ch. 4] suggests that in some cases protocols, actually make an API *easier* to use.

Similar arguments can be made about shared mutable state. The more that programmers understand the effort that goes into correctly understanding mutable state, the more likely they are to avoid it, defining immutable data structures whenever possible. The difficulties associated with using object protocols depend critically on the mutability of those objects. Each time an immutable object was encountered in our case studies, it was like a little present. The work we were required to perform was dramatically decreased. Whether or not programmers are actually trying to verify their code, understanding mutable data is much more difficult than understanding immutable data, especially when concurrency is at play. The recent resurgence in interest in functional programming seems to corroborate this view [32]. If anything, this author's experience verifying mutable data structures in concurrent programs leads him to cast his lot on the side of immutable data.

And finally, this work seems to provide some insight into a topic that has been widely studied in the past with questionable success: automatic parallelization. The thought dawned on me when I was adding synchronization to another program in my case study: "Why must I add

synchronization manually when Sync-or-Swim already knows where it needs to go?" By enriching our notion of types to the point where they contain aliasing and abstract state information, it seems that we might have provided enough information to just let the compiler sort out the details of concurrency. Access permissions encode a lot of dependency information. For example, "this object is thread-shared but immutable," and "this method cannot be called until this thread-shared object is in the 'open' state." This information forms a rich starting point for parallelization tools. Stork et al. [89] have already begun using access permissions for just such a purpose in their Æminium language. I find this project to be especially exciting.

## 9.4 Summary

This thesis was a multi-part exploration of protocols in APIs, with a particular emphasis on our ability to check their correct usage in concurrent programs.

We began with an empirical study to determine the prevalence of object protocols in the wild. This study showed that protocols were commonly defined and even more commonly used in major open-source applications.

We then showed that a particularly expressive form of access permissions [15] was a good foundation for a type system guaranteeing the absence of protocol violations in concurrent programs. The system was formalized and proved sound. This type system was later extended with parametric polymorphism, which allows programmers to specify common "generic" data structures. A static analysis for the Java programming language called, "Sync-or-Swim," was developed that incorporated many of the principles of this formal type system.

As a means of evaluating the approach, Sync-or-Swim was used to specify and verify several large open-source programs. It was found that the rate of false positives was competitive with the existing single-threaded approach, as was the specification burden. Several program bugs were found as a result of this experiment.

In order to further reduce specification burden, a system of specification inference was developed which uses probabilistic constraints. Such constraints encode specifications that are likely to be true, and in another large case study, this approach was found to perform well.

Finally, in order to provide additional motivation for programmers to use such a system, we showed that the same specifications that are used to help guarantee protocol conformance can actually be used to optimize run-time performance. To achieve this, we modified an implementation of software transactional memory to statically remove operations that the specifications indicated were unnecessary.

The effectiveness of our approach in verifying large open-source programs suggests that the approach may be helpful for real-world programmers struggling with the complex APIs in today's modern libraries and frameworks.

# Appendix A

# Examples from Each Protocol Category

In this appendix we present a code example of each protocol category. Note that for space reasons, some of these snippets have been reformatted. Their content has not been changed.

## A.1 Initialization

Package: **java.security**
Class: **AlgorithmParameters**

```
public final <T extends AlgorithmParameterSpec>
T getParameterSpec(Class<T> paramSpec)
throws InvalidParameterSpecException
{
if (this.initialized == false) {
  throw new InvalidParameterSpecException("not initialized"); //
     EVIDENCE
}
return paramSpi.engineGetParameterSpec(paramSpec);
}
```

## A.2 Boundary

Package: **java.util**
Class: **ArrayDeque.DeqIterator**

```
public E next() {
  if (cursor == fence)
    throw new NoSuchElementException(); // EVIDENCE
  E result = elements[cursor];
  // This check doesn't catch all possible comodifications,
  // but does catch the ones that corrupt traversal
  if (tail != fence || result == null)
    throw new ConcurrentModificationException();
```

```
  lastRet = cursor;
  cursor = (cursor + 1) & (elements.length - 1);
  return result;
}
```

## A.3 Deactivation

Package: **java.io**
Class: **BufferedInputStream**

```
private InputStream getInIfOpen() throws IOException {
  InputStream input = in;
  if (input == null)
    throw new IOException("Stream closed"); // EVIDENCE
  return input;
}
```

## A.4 Redundant Operation

Package: **javax.annotation.processing**
Class: **AbstractProcessor**

```
public synchronized void init(ProcessingEnvironment processingEnv) {
  if (initialized)
    throw new IllegalStateException("Cannot call init more than once."
       ); // EVIDENCE
  if (processingEnv == null)
    throw new NullPointerException("Tool provided null
       ProcessingEnvironment");

  this.processingEnv = processingEnv;
  initialized = true;
}
```

## A.5 Dynamic Preparation

Package: **java.util.concurrent**
Class: **ConcurrentLinkedQueue.Itr**

```
public void remove() {
  Node<E> l = lastRet;
  if (l == null) throw new IllegalStateException(); // EVIDENCE
  // rely on a future traversal to relink.
  l.setItem(null);
```

```
  lastRet = null;
}
```

## A.6 Type Qualifier

Package: **javax.xml.validation**
Class: **SchemaFactoryFinder.SingleIterator**

```
public final void remove() {
  throw new UnsupportedOperationException();  // EVIDENCE
}
```

## A.7 Domain Mode

Package: **javax.imageio**
Class: **ImageWriteParam**

```
public void setCompressionType(String compressionType) {
  if (!canWriteCompressed()) {
    throw new UnsupportedOperationException(
        "Compression not supported");
  }
  if (getCompressionMode() != MODE_EXPLICIT) {
    throw new IllegalStateException
        ("Compression mode not MODE_EXPLICIT!"); // CANDIDATE
  }
  String[] legalTypes = getCompressionTypes();
  if (legalTypes == null) {
    throw new UnsupportedOperationException(
        "No settable compression types");
  }
  if (compressionType != null) {
    boolean found = false;
    if (legalTypes != null) {
        for (int i = 0; i < legalTypes.length; i++) {
            if (compressionType.equals(legalTypes[i])) {
                found = true;
                break;
            }
        }
    }
    if (!found) {
        throw new IllegalArgumentException("Unknown compression type!"
           );
    }
```

```
  }
  this.compressionType = compressionType;
}
```

# Appendix B

# Proof of Safety for Single Threads

This appendix contains the complete single-threaded proof of soundness described in Chapter 4. The proofs were not put in the main body of the text for space reasons. The bulk of this appendix is made up by the proof of Theorem 3, single-threaded Progress, and Theorem 4, single-threaded Preservation, but it also contains proofs for a few smaller lemmas.

## B.1 Proof of Single-Threaded Progress

This section contains the full proof of Theorem 3.

*Proof.* By induction over the cases of the typing judgment $\Gamma; \Delta; \Psi; u \vdash \quad e : E \quad \dashv \Delta_o; u_o$.

**Case** (P-Term-I)**.**

From the syntax, $b$ can either be $x$, $l$, or $o$. If $b = x$, this would contradict the requirement that $e$ is a closed term. If $b = o$, then this is a value and the case is trivially true. If $b = l$, then by transition rule D-LOOKUP, the expression can take a step. How do we know that $l \in \text{dom}(\rho)$? The clause, $\Gamma; \rho \vdash \quad H$ ok of the heap invariant, which is given to hold, implies that $l \in \text{dom}(\rho)$ because it stipulates that for all $l \in \Gamma$, it is the case that $H(\rho(l)) = C(\overline{o})$.

**Case** (P-Term-II)**.**

This case is identical to the previous case.

**Case** (P-Load)**.**

From the truth of P-LOAD, we know that localFields$(C, f_i) = C_i$ and $l{:}C \in \Gamma$. From the truth of the heap invariant, this implies that $H(\rho(l)) = C(\overline{o})@\$$. Together this satisfies the premise of the transition rule, D-LOAD, so the expression can take a step.

**Case** (P-Assign)**.**

From the truth of P-ASSIGN, we know that localFields$(C, f_i) = C_i$ and $l{:}C \in \Gamma$. From the truth of the heap invariant, this implies that $H(\rho(l)) = C(\overline{o})@\$$. Together this satisfies the premise of the transition rule, D-ASSIGN, so the expression can take a step.

**Case** (P-Unpack-Unique)**.**

From the typing rule P-UNPACK-UNIQUE, we are given $l{:}\mathsf{unique}@S \in \Delta$. From the heap invariant, this allows us to conclude that $H(\rho(l)) = C(\overline{o})@s'$ where $s' \leftrightarrow S$. This is the premise of the transition rule, D-UNPACKUNIQ-ENTER, so the expression can take a step.

**Case** (P-InUnpack-Uniq)**.**

From the premise of the given typing rule, we know that

$$\Gamma, l{:}C; \Gamma; \Psi; \mathsf{up}(l, \mathsf{unique}, S, \mathsf{inv}_C(S)) \vdash \quad e : E \quad \dashv \Delta_o; \mathsf{up}(l, \mathsf{unique}, S, \overline{f : p}).$$

From the induction hypothesis, which is true because $e$ is well-typed and the heap invariant holds for those static contexts (given), we know that either $e$ is a value or $e$ can take a step. If $e$ can take a step, this satisfies the premise of the transition rule D-INUNPACK-UNIQ, so the entire expression can take a step. If $e$ is a value, then by the transition rule D-UNPACK-UNIQ-LEAVE the entire expression can take a step. The premise of this rule is satisfied because, by the heap invariant, if $u = \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S))$ then it is the case that $H(\rho(l)) = C(\overline{o})@\mathsf{up}$.

**Case** (P-Unpack-Imm)**.**

This case is identical to case P-Unpack-Unique above, except that for this typing rule, $k_r$ could legally be unique or immutable since they can both be split to immutable. Either way, the heap invariant guarantees that the premise of D-UNPACKIMM-ENTER holds.

**Case** (P-InUnpack-Imm)**.**

This case is identical to P-InUnpack-Uniq.

**Case** (P-Unpack-Share)**.**

This case is identical to case P-Unpack-Unique above, except that for this typing rule, $k_r$ could legally be unique or share since they can both be split to share. Either way, the heap invariant guarantees that the premise of D-UNPACKSHARE-ENTER holds.

**Case** (P-InUnpack-Share)**.**

This case is identical to P-InUnpack-Uniq.

**Case** (P-Sync)**.**

The three potentially-applicable transition rules, D-SYNC-BUSY, D-SYNC-ACQ and D-SYNC-ALREADY, describe the three possible states of the locking context $\kappa$. Either the lock is already held by another thread, it is not held by any thread or it is held by the currently executing thread. Regardless, one of the three situations will apply, so the expression can take a step.

**Case** (P-InSync)**.**

Given that the expression is typed with this typing rule, we know that the subexpression $e$ is well-typed, $\Gamma; \Delta; \Psi, l; u \vdash \quad e : E \quad \dashv \Delta_o; u_o$. Since the heap invariant is given to hold, it must also hold for these static contexts, since they are the same as used for the entire expression. This allows us to use the induction hypothesis. Either $e$ is a value or it can take a step. If it can take

a step, then this is enough to satisfy the premise of the transition rule D-INSYNC, and therefore the entire expression can take a step.

If $e$ is a value $o$, however, then one of two cases may hold. From the truth of the heap invariant, specifically the clause $\rho; \Psi; \kappa \vdash \quad \iota$ ok, we know that $\rho(l) \overset{i}{\mapsto} \iota \in \kappa$. For the value $i$, there are two possible cases. It could be equal to one, or it could be greater than one. If the former, then the rule D-SYNC-RELEASE-I applies, and the entire expression can take a step. If the latter, then rule D-SYNC-RELEASE-II applies, and the entire expression can take a step.

**Case** (P-Let)**.**

Given this typing rule, by inversion we know that $\Gamma; \Delta; \Psi; u \vdash \quad e_1 : C_1.p_1 \quad \dashv \Delta_1; u_1$. And, since the heap invariant is given to hold, it must also hold for these static typing contexts, since they are the same. Therefore, by the induction hypothesis, either $e$ is a value $o$, or it takes a step. If it takes a step, then by rule D-LET-E, the entire expression can take a step. If it is a value, then by rule D-LET-V, the entire expression can take a step.

**Case** (P-Call)**.**

For this case, there is only one applicable rule, D-CALL, and its premises are trivially satisfied by looking up the body of the method. So, the entire expression can take a step.

**Case** (P-Spawn)**.**

By rule D-SPAWN, the entire expression can take a step.

**Case** (P-New)**.**

$\mathsf{init}(C) = \langle \overline{f : C.p}, s \rangle$ is true by inversion of this typing rule. This satisfies the premise of D-NEW, so the entire expression can take a step.

□

## B.2 Proof of Single-Threaded Preservation

This section contains the full proof of Theorem 4.

*Proof.* By induction over the cases of the single-threaded step judgment,

$$(\rho; \kappa; H; \iota.e) \to (\rho'; \kappa'; H'; \iota.e'; a).$$

**Case** (D-Lookup)**.**

In this case, $l$ steps to $\rho(l)$, which we will call $o$. The expression $l$ is well-typed and it could have been typed with one of two typing rules, which we will treat as two separate sub-cases.

In the case where P-TERM-I was used to derive the typing of $l$, the following must have held: $\Gamma, l{:}C, \Delta, l{:}k@S; \Psi; u \vdash \quad l : C.k'@S' \quad \dashv \Delta, l{:}k''; u$ where $k \Rrightarrow k'/k''$ and $S \vdash \quad S'$. Therefore, choose $\Gamma' = \Gamma, l{:}C, o{:}C$ and $\Delta' = \Delta, l{:}k''@S, o{:} \downarrow (k'@S)$ and $\Psi' = \Psi, u' = u$. The resulting expression $o$ is now well-typed by rule P-TERM-II:

$$\Gamma, l{:}C, o{:}C; \Delta, l{:}k''@S, o{:} \downarrow (k'@S); \Psi; u \vdash \quad o : C. \downarrow (k'@S') \quad \dashv \Delta, l{:}k''@S; u$$

We can show that each of the side-conditions hold:

- $\downarrow (C.k'@S') = \downarrow (C.k'@S')$
- $\Delta'_o = \Delta_o$
- $u'_o = u_o$
- $\Gamma, l{:}C \leq \Gamma, l{:}C, o{:}C$
- True, since $\Psi = \Psi'$ and $\mathsf{activeLocks}(e) = \mathsf{activeLocks}(e') = \emptyset$.
- Applies, but is true, since $u' = u$.

If, alternatively, the rule P-TERM-II were used to derive the type of $l$, then the following would have held: $\Gamma, l{:}C; \Delta, l{:}k@S; \Psi; u \vdash \quad l : C.k@S' \quad \dashv \Delta$ where $S \vdash \quad S'$. So choose $\Gamma' = \Gamma, l{:}C, o{:}C$ and $\Delta' = \Delta, \downarrow (o{:}k@S)$ and $\Psi' = \Psi, u' = u$. Now again use P-TERM-II to show that the expression is well-typed:

$$\Gamma, l{:}C, o{:}C; \Delta, o{:} \downarrow (k@S); \Psi; u \vdash \quad o : C. \downarrow (k'@S') \quad \dashv \Delta; u$$

And again, the side-conditions hold, for the same reasons:

- $\downarrow (C.k@S') = \downarrow (C.k@S')$
- $\Delta'_o = \Delta_o$
- $u'_o = u_o$
- $\Gamma, l{:}C \leq \Gamma, l{:}C, o{:}C$
- True, since $\Psi = \Psi'$ and $\mathsf{activeLocks}(e) = \mathsf{activeLocks}(e') = \emptyset$.
- Applies, but is true, since $u' = u$.

$\mathsf{stackWF}(\Psi', u', e', \rho', \kappa', \iota)$ is well-formed because it held before the step, $\Psi$ and $\kappa$ have not changed, and $\mathsf{numLocks}(e) = \mathsf{numLocks}(e') = 0$. The $\mathsf{upProtect}$ clause holds because it held before the step, $u = u'$ and $\Psi$ has not changed.

The single-threaded guarantees are satisfied. By this rule some permission may be dropped from $\Delta$ but not moved into any other context. The heap was not changed in any way.

However, we also must show that the heap invariant holds. Because the heap invariant was true before the step, and $H(\rho(l)) = C(\overline{o})@\$$, $H(o) = C(\overline{o})@\$$ is true after so $\Gamma'$ is consistent. Similarly, $\Delta'$ is consistent because it shares a permission that was true before the step. $u$ has not changed, so it remains consistent, along with $\mathsf{shareUnpack}$. Finally, $\mathsf{shareLocks}$ is somewhat interesting. If $k$ was initially $\mathsf{share}$ then the downgrade operation performed on $o$'s permission will ensure that $S'$ is not specific.

**Case** (D-InSync)**.**

By the only applicable typing rule, P-INSYNC, the well-typedness of the expression requires that the subexpression is also well-typed, $\Gamma; \Delta; \Psi, l; u \vdash \quad e : E \quad \dashv \Delta_o; u_o$. And since we know that the heap invariant holds for these contexts, the induction hypothesis tells us that when $e$ steps to $e'$, the following will hold: $\Gamma'; \Delta'; \Psi'; u' \vdash \quad e' : E' \quad \dashv \Delta'_o; u'_o$, along with the heap invariant, $\Gamma'; \Delta'; \Psi'; u' \vdash \quad \iota; \rho'; \kappa'; H'$.

Choose theses static contexts to type $\texttt{insync}(l)\ e'$. We would like to type the entire expression by rule P-INSYNC, $\Gamma'; \Delta'; \Psi'; u' \vdash \quad \texttt{insync}(l)\ e' : E' \quad \dashv \Delta'_o; u'_o$. But to do so we must show that $l \in \Psi'$.

From $\mathsf{stackWF}$, we know that before the step $\mathsf{occurrences}(\Psi, l) \geq \mathsf{numLocks}(\texttt{insync}(l)\ e)$. We also know that $\mathsf{numLocks}(\texttt{insync}(l)\ e)$ must be at least one by its definition. But is it greater

than one? Consider both cases. If it is one, then this means $l \notin \mathsf{activeLocks}(e)$. This means from the I.H., $e$ cannot remove $l$ from $\Psi$ when it steps, so it must remain in $\Psi'$. If, on the other hand, $\mathsf{numLocks}$ is greater than one for the entire expression, it must be because $l \in \mathsf{activeLocks}(e)$. $\Psi$ must contain one more entry $l$ than the $\mathsf{numLocks}(e)$, and because by the I.H., it can only remove the number of copies from $\Psi$ corresponding to $\mathsf{numLocks}(e, l)$, there must be at least one remaining entry $l$ after the step.

All of the side-conditions hold by the induction hypothesis.

The stack well-formedness property continues to hold from the induction hypothesis. Again, the subexpression can remove lock counts and remove locks from $\Psi$, but it is bounded by the number of active locks it contains, and the entire expression will always contain one more.

The single-threaded guarantees are satisfied by the induction hypothesis.

If $a = e_2$, then the conditions on newly spawned threads hold by a straightforward application of the induction hypothesis.

Since we are using all of the original typing contexts, the heap invariant holds from the induction hypothesis.

**Case** (D-InUnpack-Uniq)**.**

By the only typing rule that could have been used in this case, P-INUNPACK-UNIQ, we know that $e$ must be well-typed, $\Gamma, l : C; \Delta; \Psi; \mathsf{up}(l, \mathsf{unique}, S, \overline{f : p}) \vdash \quad e : E \quad \dashv \Delta_o; u_o$, and since we are given that the heap invariant is satisfied for these contexts, we know that $\Gamma'; \Delta'; \Psi'; u' \vdash \quad e' : E' \quad \dashv \Delta'_o; u'_o$, and that these contexts satisfy the new heap run-time state, by the induction hypothesis. We would like to use the typing rule P-INUNPACK-UNIQ to show that the whole expression is well-typed. To do this, first we must show that $l : C \in \Gamma'$, but this is given to us by the induction hypothesis, since $\Gamma' \geq (\Gamma, l : C)$. Similarly, we must show that $\overline{f' : p'} \leftrightarrow \mathsf{inv}_C(s)$, but this is also given to us by the induction hypothesis which says that $u'_o \geq u_o$.

Finally, we must show that $u'$ has the form $\mathsf{up}(l; \mathsf{unique}; S; \overline{f' : p'})$. From $\mathsf{stackWF}$, we know that $\mathsf{activeUnpack}(\texttt{inunpackuniq}(l)\ e)$ is defined. This in turn means that $\mathsf{activeUnpack}(e) = \emptyset$, from the definition of $\mathsf{activeUnpack}$. So, by the preservation side-condition, we know that $u' = \mathsf{up}(l; k; S; \overline{f' : p'})$.

The stack is still well-formed. $\Psi', \kappa'$ and $\mathsf{numLocks}(e')$ are all consistent with one another by the induction hypothesis. The locks in $\kappa$ are okay by the I.H. The same is true for $\mathsf{activeLocks}$ being defined. We know that $\mathsf{activeUnpack}$ is defined because the side-condition on the I.H. says that $\mathsf{activeUnpack}(e') = \emptyset$, so by definition $\mathsf{activeUnpack}(\texttt{inunpackuniq}(l)\ e') = \{l\}$. $\mathsf{upProtect}$ is trivially true, since $u' = \mathsf{up}(l; \mathsf{unique}; S; \overline{f' : p'})$.

The single-threaded guarantees are satisfied from the induction hypothesis.

The side-conditions still hold:

- By I.H.
- By I.H.
- By I.H.
- By I.H.
- By I.H.
- $\mathsf{activeUnpack}(\texttt{inunpackuniq}(l)\ e) \neq \emptyset$

The heap invariants also hold, all of them by the I.H.

If $a = e_2$, then the conditions on newly spawned threads hold by a straightforward application of the induction hypothesis.

**Case** (D-InUnpack-Imm)**.**

Identical to the previous case, except using typing rule P-INUNPACK-IMM.

**Case** (D-InUnpack-Share)**.**

Identical to the previous case, except using typing rule P-INUNPACK-SHARE.

**Case** (D-Let-E)**.**

By the only applicable typing rule, P-LET, we know that both subexpressions are well-typed, in particular, $\Gamma; \Delta; \Psi; u \vdash \ e_1 : C_1.p_1 \ \dashv \Delta_1; u_1$. Because we are given that the heap invariant holds for these static contexts with the current run-time state, we can use the induction hypothesis to show that the expression to which it steps is well-typed, $\Gamma'; \Delta'; \Psi'; u' \vdash \ e_1' : C_1.p_1' \ \dashv \Delta_1'; u_1'$. Moreover, the heap invariant for those contexts holds. Because the typing rule held before the step, we know that $\Psi = \mathsf{activeLocks}(e_1), \Psi_2$. And, from the induction hypothesis, we know that $\Psi' = \mathsf{activeLocks}(e_1'), \Psi_2$.

In order to use the typing rule P-LET to type the new expression, we must show that the following holds: $\Gamma', x{:}C_1; \downarrow^{\Psi_2} (\Delta_1'), x{:} \downarrow (p_1); \Psi_2; u' \vdash \ e_2 : E_2' \ \dashv \Delta_s'; u_s'$. We have a number of different weakening lemmas that makes this so.

By the I.H. we know $\Gamma \leq \Gamma'$ and because $\Gamma, x{:}C_1 \leq \Gamma', x{:}C_1$, we know $e_2$ has the same type under this new larger context. Similarly, the I.H. says that $u_1 \leq u_1'$ and by the Unpacking Weakening lemma, $e_2$ has the same type under this new packing flag, with the exception of a new output packing flag $u_o'$ such that $u_o \leq u_o'$. $\Psi_2$ was previously used to type $e_2$ and has not changed.

The linear context is interesting. By I.H., when $e_1$ takes a step we have a new output linear context $\Delta_1'$ for which it must be the case that $\downarrow^{\Psi_2} (\Delta_1) \leq \downarrow^{\Psi_2} (\Delta_1')$. Also, $\downarrow (p_1) = \downarrow (p_1')$, by the I.H., so we can conclude that $\downarrow^{\Psi_2} (\Delta_1), x{:} \downarrow (p_1) \leq \downarrow^{\Psi_2} (\Delta_1'), x{:} \downarrow (p_1')$. By the weakening lemma for linear contexts, $e_2$ remains well-typed, under this larger context with the caveat that $\Delta_o \leq \Delta_o'$.

The stack is well-formed. This follows from the induction hypothesis and because a `let` expression modifies neither activeLocks nor activeUnpack.

The single-threaded guarantees are satisfied from the induction hypothesis.

The side-conditions hold:

- The types are the same.
- From the Linear Weakening lemma
- From the Unpacking Weakening lemma
- From the I.H.
- From the I.H.
- From the I.H. and the fact that the second expression $e_2$ is not typed under $u'$ but rather $u_o'$ for which this does not apply.

We need to show that the heap invariant holds for the new contexts and run-time state. However, all of the premises of the heap invariant hold by the induction hypothesis.

We do not know whether or not $a = e_s$, but if it does, then by the induction hypothesis, we can split $\Delta$ such that $e_s$ is well-typed. But more importantly in this case, because the output linear context $\Delta_o$ and the output packing flag $u_o$ must be exactly the same under this new linear context $\Delta_1$, expression $e_2$ remains well-typed.

**Case** (D-Let-V)**.**

By the one typing rule that could have been used to derive this expression, P-LET, we know that $\Gamma; \Delta; \Psi; u \vdash \ o : C_1.p_1 \ \dashv \Delta_1; u_1$ and $\Gamma, x{:}C_1, {\downarrow^{\Psi}}(\Delta_1), x{:}\downarrow(p_1); \Psi; u_1 \vdash \ e : E \ \dashv \Delta_o; u_o$. For our final static contexts, choose $\Gamma' = \Gamma, l{:}C_1$ and $\Psi' = [l/x]\Psi$ and $\Delta' = {\downarrow^{\Psi'}}(\Delta_1), l{:}\downarrow(p_1)$ and $u' = [l/x]u_1$.

It important to note that $\Psi' = \Psi$ and $u' = u_1$, since neither context could contain $x$.

By Lemma 2, our substitution lemma, these contexts can be used to type $e'$ (i.e., $\Gamma', \Delta'; \Psi'; u' \vdash \ [x/l]e : E' \ \dashv \Delta'_o; u'_o$. This means that P-LET can be used to type the entire expression.

The stack is still well-formed, since we have not modified the active locks or the actively unpacked expression in any way. Same goes for $\Psi$ and $\kappa$. This is true for the upProtect clause because we know that $u' = u_1 = u$ (from the only typing rules that can be used on $o$), and since $\kappa$ has not changed, and $\mathsf{activeLocks}(e) = \mathsf{activeLocks}(e') = \emptyset$.

The single-threaded guarantees are satisfied. This rule has not actually moved any permission. By the only typing rules applicable for $o$, we know that $u' = u$. And while $\Delta'$ is different from $\Delta$, permission has only been changed to point to the object label $l$ from the object itself. The heap has not been changed.

The side-conditions continue to hold:

- Given by substitution lemma, $E$ does not change.
- Original $\Delta_o$ did not contain $x$, so true by substitution lemma.
- Original $u_o$ did not contain $x$, so true by substitution lemma.
- Yes, since we have just added $l$.
- Yes. $\Psi_2 = \Psi' = \Psi$.
- Yes, since we have not modified the active unpack expression in any way, nor modified the packing flag significantly.

The heap invariant also continues to hold. The first premise ensures that the types in $\Gamma'$ reflect the types in the heap. For the only new entry added to $\Gamma'$, we know $H'(\rho'(l)) = C(\overline{o})@\$$ because $H(o) = C(\overline{o})@\$$ was true before the step and $\rho'(l) = o$. Similarly for $\Delta'$ we know that $\Delta, l{:}k@S; \rho' \vdash \ H'$ because before the step $\Delta, o{:}k@S; \rho \vdash \ H'$ and $\rho'(l) = o$. For the locking premise, we know that $\Psi = \Psi'$. It also could not have contained $o$ since syntactically this is impossible. So, since this premise held before the step it continues to hold.

How do we know the new packing flag $u_1$ is consistent with the heap? In fact, we know a lot about the shape of $u_1$. First, it cannot contain $x$, since $x$ was not in scope, so $[x/l]u_1 = u_1$. Furthermore, by examining the only two typing rules that could have been used to type $o$, we know that $u_1$ must be equal to $u$. This means that $u_1$ is consistent with the heap because $u$ was consistent with the heap before the step. shareLocks is okay because our new linear context is

a down-grading of the permissions for which we do not have locks. And again, shareUnpack must hold because $u = u'$ and $u$ was consistent with the heap before the step.

**Case** (D-Load)**.**

By the only typing rule that could be used to type a field dereference, we know that this expression was well typed, $\Gamma, l{:}C; \Delta; \Psi; u \vdash \quad l.f_i^k \ : \ C_i.k@S_i \quad \dashv \ \Delta; u_o$, where $u = \mathsf{up}(l; k_r; S_r; \overline{f : p}, f_i : k_i@S_i)$ and $k_i \Rrightarrow k/k_i'$ and $u_o = \mathsf{up}(l; k_r; S_e; \overline{f : p}, f_i : k_i'@S_i)$.

Choose $\Gamma' = \Gamma, l{:}C, o_i{:}C_i$ and $\Delta' = \Delta, o_i{:}k@S_i$ and $\Psi' = \Psi$ and $u' = \mathsf{up}(l; k_r; \overline{f : p}, f_i{:}k_i'@S)$. By typing rule P-TERM-II we can derive $\Gamma'; \Delta'; \Psi'; u' \vdash \quad o_i : C_i.k@S \quad \dashv \Delta; u'$.

The stack is still well-formed because we have not changed the active locks or unpacks in any way. Same goes for $\Psi$ and $\kappa$. This includes the upProtect clause which, if $k_r = \mathsf{share}$, will hold because $\kappa$ and activeLocks have not changed.

The single-threaded guarantees are satisfied. Permission for $o$ has been moved from the packing flag to the new linear context. The heap has not been changed, so permsNeeded holds.

The side-conditions hold:

- $C_i.k@S = C_i.k@S$
- The outgoing linear contexts are the same before and after.
- The outgoing packing contexts are the same before and after.
- $\Gamma$ has only been increased, to add $o{:}C_i$.
- True, since $\Psi' = \Psi$.
- This is true for $e$ and $e'$.

Finally, the heap invariant holds. From the heap invariant itself we know that $H(\rho(l)) = C(\overline{o})$ implies $H(o_i) = C_i(\overline{o})$. $\Psi$ was correct with respect to the locking state before, so our new $\Psi' = \Psi$ is also correct, since $\kappa$ has not changed. $\Delta, o_i{:}k@S$ is consistent with the heap because $\Delta$ was consistent with the heap before the step, the heap has not changed, and since the invariant defines consistency between the packing flag and the heap, we know $o_i{:}k@S_i; \rho \vdash \quad H$. Consistency between $u'$ and our heap holds, because only one entry was changed, the entry for $f_i$, and the state is the same. shareLocks is okay because no precise share permission can ever be in $u$ in a well-typed program (by Lemma 9). Finally, if shareUnpack was true before the step it must be true afterward.

**Case** (D-Assign)**.**

By the only typing rule that could have been use to type a field assignment, P-ASSIGN, we know how the original expression must have been typed, $\Gamma, l_1{:}C; \Delta; \Psi; u \vdash \quad l_1.f_i := l_2 : C_i.k_i@S_i \quad \dashv \Delta_o; u_o$, where $u = \mathsf{up}(l_1; k_1; S_1; \overline{f : p}, f_i : k_i@S_i$. From the premises of this typing rule, we know that the assigned label is well-typed, $\Gamma, l_1{:}C; \Delta; \Psi; u \vdash \quad l_2 : \underline{C_i.p} \quad \dashv \Delta_o; u$ and that the outgoing packing context has the following form, $u_o = \mathsf{up}(l_1; k_1; S_1; \overline{f : p}, f_i :\downarrow (p))$.

So, choose $\Gamma' = \Gamma, l_1{:}C, o_i{:}C_i$ and $\Delta' = \Delta, o_i{:}k_i@S_i$ and $\Psi' = \Psi$ and $u' = \mathsf{up}(l_1; k_1; S_1; \overline{f : p}, f_i \ :\downarrow \ (p))$. By P-TERM-II, the resulting expression is well-typed, $\Gamma'; \Delta'; \Psi'; u' \vdash \quad o_i : C_i.k_i@S_i \quad \dashv \Delta; u'$.

The stack well-formedness property still holds because we have not changed anything about the active packing or locking expressions. Same goes for $\Psi$ and $\kappa$. This includes the upProtect clause which, if $k_r = \mathsf{share}$, will hold because $\kappa$ and activeLocks have not changed.

The single-threaded guarantees are satisfied. Some permission has been moved from the packing flag to the linear context and vice-versa. The field of $l_1$ has been changed, but we know $k_1$ is a writing permission, so permsNeeded is satisfied.

The side-conditions hold:

- $C_i.k_i@S_i = C_i.k_i@S_i$
- The outgoing linear contexts are the same before and after.
- The outgoing packing contexts are the same before and after.
- The new $\Gamma'$ is the old $\Gamma$ with an additional entry for $o$, so it is larger.
- True, since $\Psi' = \Psi$.
- This condition is relevant and holds.

Finally, we claim that the heap invariant holds. From the heap invariant itself we know that $H(\rho(l)) = C(\overline{o})$ implies $H(o_i) = C_i(\overline{o})$. $\Psi$ was correct with respect to the locking state before, so our new $\Psi' = \Psi$ is also correct, since $\kappa$ has not changed. $\Delta, o_i{:}k_i@S_i$ is consistent with the heap because $\Delta$ was consistent with the heap before the step, the heap has not changed, and by the definition of consistency between the packing flag and the heap, $o_i{:}k_i@S_i; \rho \vdash \ H$.

The most interesting question is why the packing flag is consistent with the heap now that we have changed it. We must show $l_2{:}\downarrow(p); \rho \vdash \ H$. By the only typing rules that could have been used to type $l_2$, we can see that consistency between $\Delta$ and the heap implies $l_2{:}\downarrow(p); \rho \vdash \ H$, particularly when we take into account the fact that $p$ will be downgraded if it is a share permission.

**Case** (D-Sync-Busy)**.**

This case is rather simple, since the expression steps back to the exact same expression. The same typing rule that was used to type the expression, P-SYNC is again used to type it after its step, with all the same static contexts.

The stack well-formedness lemma still holds because neither the expression nor the heap has changed. Same goes for the single-threaded guarantees and the heap invariant.

**Case** (D-Sync-Acq)**.**

By the only typing rule that could have been used to type the expression before the step, we know, $\Gamma, l{:}C; \Delta; \Psi; u \vdash \ \texttt{synchronized}\ (l)\ e : E \ \dashv \Delta_o; u_o$. And by inverting this rule, we also know that the subexpression $e$ is well-typed, $\Gamma, l{:}C; \Delta; \Psi, l; u \vdash \ e : E \ \dashv \Delta_o; u_o$. This is enough to satisfy the premise of the P-INSYNC typing rule, so we can say $\Gamma, l{:}C; \Delta; \Psi, l; u \vdash \ \texttt{insync}(l)\ e : E \ \dashv \Delta_o; u_o$.

The stack is well-formed. activeLocks was defined before the step, and in order for activeLocks($\texttt{synchronized}\ (l)\ e$) to be defined, activeLocks$(e) = \emptyset$. It remains well-defined after the step, and in fact activeLocks($\texttt{insync}(l)\ e$) $= \{l\}$. Because before the step this lock was not in $\kappa$, we know that numLocks($\texttt{synchronized}\ (l)\ e, l$) $=$ occurrences$(\Psi, l) =$ occurrences$(\Psi, \rho, o) = 0$ and numLocks($\texttt{synchronized}\ (l)\ e, \rho, o$) $=$ occurrences$(\Psi, \rho, o) = 0$. Now that $i$ is incremented to 1, these relationships still hold: numLocks($\texttt{insync}(l)\ e, l$) $=$ occurrences$(\Psi', l) =$ occurrences$(\Psi', \rho', o) = 1$ and numLocks($\texttt{insync}(l)\ e, \rho', o$) $=$ occurrences$(\Psi', \rho', o) = 1$.

upProtect also holds after the step because, while activeLocks$(e')$ has increased by one, the count in $\kappa'$ has also increased by one.

The single thread guarantees are satisfied. The permission contexts, packing flag and heap have not changed.

The side-conditions hold.

- $E = E'$
- $\Delta_o = \Delta'_o$
- $u_o = u'_o$
- $\Gamma = \Gamma'$
- By the definition of activeLocks(`synchronized` $(l)$ $e$) it must be the case that activeLocks$(e) = \emptyset$. So, before the step, $\Psi_2 = \Psi$. After the step, $\Psi' = \Psi, l = \Psi_2,$ activeLocks$(e')$.
- This was true before the step and we have not changed $u$ or the active unpack state of the expression.

Finally, the heap invariant holds. $\Gamma$ has not changed and thus is still consistent with the heap. We added $l$ to $\Psi$ but we know $\rho(l) \overset{1}{\mapsto} \iota \in \kappa'$ from the premise of D-SYNC-ACQ. $\Delta$ has not changed and the unpacking flag has not changed, so they are still consistent with the heap. The shareUnpack and shareLocks premises still hold because we have only added locks to the locking context. Any share permissions that needed protection are still protected under the new context.

**Case** (D-Sync-Already)**.**

The new expression is well-typed for exactly the same reasons as in the previous case. The only things that may differ are the truth of the stack well-formedness and heap invariant.

The stack is well-formed. activeLocks was defined before the step, and in order for activeLocks(`synchronized` $(l)$ $e$) to be defined, activeLocks$(e) = \emptyset$. It remains well-defined after the step, and in fact activeLocks(`insync`$(l)$ $e) = \{l\}$.

The relationships between the number of occurrences of $l$ in $\Psi$ and the number of locks in the expression held before the step. The continue to hold after the step because the value for each has increased by one:

$$\begin{gathered}\mathsf{numLocks}(\texttt{insync}(l)\ e, l) = \mathsf{numLocks}(\texttt{synchronized}\ (l)\ e, l) + 1 \\ \mathsf{occurrences}((\Psi, l), l) = \mathsf{occurrences}(\Psi, l) + 1 \\ \mathsf{occurrences}((\Psi, l), \rho, o) = \mathsf{occurrences}(\Psi, \rho, o) + 1 \\ \mathsf{numLocks}(\texttt{insync}(l)\ e, \rho, o) = \mathsf{numLocks}(\texttt{synchronized}\ (l)\ e, \rho, o) + 1\end{gathered}$$

The upProtect clause also holds after the step, since the size of activeLocks$(e')$ has increased by one but the lock count in $\kappa'$ has also increased by one.

The single-thread guarantees are satisfied because the permission context, the packing context and the heap have not changed.

The heap invariant holds. Similar to in the previous case, the only thing we really need to worry about is the consistency between $\Psi'$, which now contains $l$, and $\kappa'$. However, from the premise of the step rule, D-SYNC-ALREADY, we know that $\rho(l) \overset{i+1}{\mapsto} \iota \in \kappa'$, so the heap invariant is satisfied.

**Case** (D-Sync-Release-I)**.**

By inverting the only typing rule that could be used to type this expression, P-INSYNC, we know that $o$ is well-typed, $\Gamma; \Delta; \Psi, l; u \vdash \ o : E \ \dashv \Delta_o; u_o$. By Lemma 8, because $o$ is a value, it must be the case that $\Gamma; \Delta; \Psi, l; u \vdash \ o : E \ \dashv \Delta_o; u_o$. But, we want to choose our new $\Delta'$ to be $\downarrow^{\Psi}(\Delta)$. (And note that in this case, $\Psi_2 = \Psi$.) The expression is still well-typed under the new linear context, $\Gamma; \downarrow^{\Psi}(\Delta); \Psi, l; u \vdash \ o : E \ \dashv \Delta''_o; u_o$ because, syntactically $\Psi$ cannot hold a lock for $o$ and therefore nothing in $\Delta$ can be forgotten that had not already been forgotten before the step.

The stack is well-formed. $\mathsf{activeLocks}$ is still defined after the step.

To show that the relationships between the number of occurrences of locks in $\Psi'$ are equal to zero, we use a fact from before the step. Before the step, we know that $\mathsf{occurrences}((\Psi, l), \rho, o) = 1$, since $i = 1$ is in the premise of this transition rule. By removing $l$ from $\Psi$ to get $\Psi'$, it must now be the case that $\mathsf{occurrences}(\Psi, \rho, o) = 0$, along with $\mathsf{occurrences}(\Psi, l) = 0$.

The $\mathsf{upProtect}$ clause holds after the step because the size of $\mathsf{activeLocks}(e')$ has decreased by one, and the lock count in $\kappa'$ has decreased by one.

The single-thread guarantees are satisfied because the permission context, the packing context and the heap have not changed.

The side-conditions hold.

- $E = E'$ here
- We must show that $\downarrow^{\Psi_2}(\Delta_o) \leq \downarrow^{\Psi_2}(\Delta'_o)$. In this case, $\Psi_2 = \Psi$. By examining the only two typing rules that can be used to type $o : E$ we know that the outgoing linear context $\Delta_o$ is equal to the incoming context $\Delta$ with the sole exception of $o$. Downgrading $\Delta$ with $\Psi$ cannot affect $o$ since $\Psi$ can only contain $l$ syntactically, so $\downarrow^{\Psi}(\Delta)$ must mean that $\Delta''_o = \downarrow^{\Psi}(\Delta_o)$, and $\downarrow^{\Psi}(\Delta_o) \leq \downarrow^{\Psi}(\Delta_o)$.
- $u_o = u'_o$
- $\Gamma = \Gamma'$
- Before the step $\Psi = \Psi_2$, since $\Psi, l = \Psi, \mathsf{activeLocks}(e)$. Now, $\Psi' = \Psi = \Psi_2, \mathsf{activeLocks}(e')$.
- Since we do not modify the packing flag nor whether the new expression is active unpack, this implication holds.

Finally, the heap invariant holds. The most interesting part is showing that precise $\mathsf{share}$ permissions in $\Delta'$ are still protected, along with the packing flag if it is unpacked with $\mathsf{share}$ permission. Since we chose $\Delta'$ to forget all $\mathsf{share}$ precise permissions not protected by $l$, we know that the new linear context is consistent.

$\Psi'$ is consistent with the held locks, because we have shown that $\mathsf{occurrences}(\Psi', \rho, o) = 0$.

We can show that $u$ cannot be $\mathsf{up}(l; \mathsf{share}; S; \overline{f : p})$, so $\mathsf{unpackShare}$ is trivially true. We know that $\rho(l) \overset{1}{\mapsto} \ \in \kappa$. We also know that the $\mathsf{upProtect}$ clause of the stack well-formedness property held before the step. This says either $\mathsf{nestingShare}(e)$ or $u = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p}) \Rightarrow \rho(l) \overset{i}{\mapsto} \ \in \kappa \wedge i > \mathsf{activeLocks}(e)$. But, since $\mathsf{nestingShare}(e)$ does not hold, and $i = 1 = \mathsf{activeLocks}(e)$, it cannot be the case that $u = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p})$.

**Case** (D-Sync-Release-II)**.**

This case is identical to the previous case. This time, however it is possible that $u' = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p})$. In this case, we know that the original $\Psi$ must contain an additional $l$ from the upProtect clause of the stack well-formedness predicate. This means that shareUnpack from the heap invariant will hold.

**Case** (D-Unpackuniq-Enter)**.**

Since there is only one applicable typing rule, P-UNPACK-UNIQUE, we know that the expression is well-typed $\Gamma, l{:}C; \Delta, l : \mathsf{unique}@S; \Psi; \mathsf{p} \vdash \texttt{unpackuniq}(l, S, s)\ e : E \dashv \Delta_o, l : \mathsf{unique}@s; \mathsf{p}$. And from inverting this typing rule, we know that the subexpression is also well-typed, $\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S)) \vdash e : E \dashv \Delta_o; \mathsf{up}(l; \mathsf{unique}; S; \overline{f' : p'})$. Where additionally, $\overline{f' : p} \leftrightarrow \mathsf{inv}_C(s)$. This is enough to satisfy the premises of the typing rule P-INUNPACK-UNIQ, and give us $\Gamma, l{:}C; \Delta; \Psi; \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S)) \vdash \texttt{inunpackuniq}(l; S; s)\ e : E \dashv \Delta_o, l : \mathsf{unique}@s; \mathsf{p}$.

The stack well-formedness property holds. Active unpack is defined for the new expression, $\mathsf{activeUnpack}(\texttt{inunpackuniq}(l, S, s)\ e) = \{l\}$ (since we knew $\mathsf{activeUnpack}(e) = \emptyset$ before the step). Since $\Psi'$ and $\kappa'$ are the same as they were before the step, the lock restrictions must also hold. The clause upProtect is trivially true since $u' = \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S))$.

The single-thread guarantees are satisfied. Permission has been removed from the linear context $\Delta$, and state invariant permission has been moved from the heap to the packing flag. This is allowed. The state of an object in the heap was changed, but permission to that object was present in $\Delta$ before the step, so permsNeeded is satisfied.

The side-conditions hold:

- The types are the same before and after.
- The output linear contexts are the same before and after.
- The output unpacking flag is the same before and after.
- The valid context is the same before and after.
- From the definition of activeLocks, before the step there were no active locks in the expression, so $\Psi = \Psi_2$. After the step, there are still no active locks, and $\Psi' = \Psi_2 = \Psi$.
- This implication does not apply since $u = \mathsf{p}$.

Finally, the heap invariant holds. The only interesting change is that the new packing flag, $u' = \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S))$, must be consistent with the heap. This requires that $H'(\rho'(l)) = C(\overline{o})@\mathsf{up}$, which is true from the premise of the transition rule D-UNPACKUNIQ-ENTER. Additionally, the states of the fields as listed in the packing flag must be consistent with the heap. But we know this is true because before the step, $\mathsf{inv}_C(S)$ was consistent with the heap from the consistency of $\Delta$ and the heap. What if there were other permissions in $\Delta$ that point to the same object? They would not be consistent with the heap after this step. Fortunately, the permsConsistent clause of the heap invariant before the step guarantees us that there cannot be any other permissions pointing to the same object.

**Case** (D-Unpackimm-Enter)**.**

This case is largely the same as the previous case.

**Case** (D-Unpackshare-Enter)**.**

This case is largely the same as the previous case. However, instead of choosing $\Delta$ as our new linear context, we will choose $\downarrow(\Delta)$. This is important as it will ensure that any specific share permissions in the linear context that point to the same $o$ that $l$ points to are downgraded. This in turn means that the heap invariant will be satisfied, since normally a specific share requires the object in the heap to be in a state that it now will no longer be in.

In the heap invariant, shareUnpack holds for our new packing flag because $l \in \Psi'$.

Also, the upProtect clause of the stack well-formedness property is interesting. It holds after the step because $\mathsf{activeLocks}(\texttt{inunpackshare}(l)\ e) = \mathsf{activeLocks}(e) = \emptyset$ which we know because $\mathsf{activeLocks}(\texttt{unpackshare}(l)\ e) = \emptyset$ and requires that $\mathsf{activeLocks}(e) = \emptyset$. Moreover, since $l \in \Psi'$, we know that the flag in $\kappa'$ must be at least one.

**Case** (D-Unpack-Uniq-Leave)**.**

By inversion of the only applicable typing rule, P-INUNPACK-UNIQ, we know that $o$ is well-typed under the following contexts: $\Gamma, l : C; \Delta; \Psi; \mathsf{up}(l; \mathsf{unique}; S; \overline{f : p}) \vdash \quad o : E \quad \dashv \Delta_o; \mathsf{up}(l; \mathsf{unique}; S; \overline{f' : p'})$ where $\overline{f' : p'} \leftrightarrow \mathsf{inv}_C(s)$. Choose $\Delta' = \Delta, l{:}\mathsf{unique}@s$. $\Gamma, l : C; \Delta'; \Psi; \mathsf{up}(l; \mathsf{unique}; S; \overline{f : p}) \vdash \quad o : E \quad \dashv \Delta'_o; \mathsf{up}(l; \mathsf{unique}; S; \overline{f' : p'})$ is well-typed by the weakening lemma for the linear context (Lemma 4). Choose $u' = \mathsf{p}$. $\Gamma, l : C; \Delta'; \Psi; \mathsf{p} \vdash \quad o : E \quad \dashv \Delta'_o; \mathsf{p}$ since the only typing rules that can be used for an expression $o$ ignore the packing context completely. This is our new, well-typed expression.

The stack well-formedness property holds. activeUnpack and activeLocks are still defined, but the number of active locks has stayed at zero, and otherwise $\Psi$ and $\kappa$ have not changed. upProtect is trivially true since $u' = \mathsf{p}$.

The single-thread guarantees are satisfied. There is a new permission in $\Delta'$ but it comes from a newly packed object. The state of an object in the heap has also been changed, but that is allowed by permsNeeded because the packing flag has changed from an unpacked object to a packed object.

The side-conditions all hold:

- $E = E'$
- True by the linear weakening lemma.
- True because $u = \mathsf{p} = u'$.
- True, because $\Psi = \Psi'$ and $\mathsf{activeLocks}(e) = \mathsf{activeLocks}(e') = \emptyset$.
- Trivially true since it does not apply.

The heap invariant also holds. The valid context is the same so it remains consistent with $H'$ (which is equal to $H$). All of the premises hold as unchanged except for, $\Delta, l{:}\mathsf{unique}@s; \rho \vdash \quad H'$. $H'(\rho'(l)) = C(\overline{o})@s$ by the premise of D-UNPACK-UNIQ-LEAVE. From the fact that the heap invariant was true before the step, we know that $\mathsf{up}(l; \mathsf{unique}; S; \overline{f : p})$ is consistent with the heap, meaning that all permissions $p_i$ accurately describe their objects. But, by the only two typing rules that could possibly have been use to type $o$, we know that $u = u'$, which means that $\overline{f : p} \vdash \quad \mathsf{inv}_C(s)$. This satisfies the final requirement, that $\overline{o : p}; \rho' \vdash \quad H'$.

**Case** (D-Unpack-Imm-Leave)**.**

This case is quite similar to the previous one. The main difference is that now we must show that the new linear context, $\Delta, l{:}\mathsf{immutable}@s$ is sound with respect to the heap. We can do this because of our heap invariant. It says that before the step, because $u = \mathsf{up}(l; \mathsf{immutable}; s; \overline{f : p})$, the heap must contain $H(\rho(l)) = C(\overline{s})@\mathsf{ro}(s')$ where $s \leftrightarrow s'$. This means by definition that $s = s'$. This same condition is necessary in order for $\Delta, l{:}\mathsf{immutable}@s$ to be a consistent linear context.

**Case** (D-Unpack-Share-Leave)**.**

This case is quite similar to the previous one.

**Case** (D-Call)**.**

By the only applicable typing rule, P-CALL we know that the expression must be well-typed, $\Gamma; \Delta; \Psi; \mathsf{p} \vdash \quad l.m(\bar{l}) : C_r.p_r \quad \dashv \Delta_o, l{:}p', \overline{l{:}p'}; \mathsf{p}$. And by inverting that rule we also know that $\Gamma; \Delta; \Psi; \mathsf{p} \vdash \quad l : C.p, \overline{l : C.p} \quad \dashv \Delta_o$.

From the rule that was used to type each method declaration, P-METH-DECL, we know that $\mathtt{this}{:}C, \overline{x{:}C}; \mathtt{this}{:}p, \overline{x{:}p}; \bullet; \mathsf{p} \vdash \quad e : C_r.p_r \quad \dashv \Delta'_o, \mathtt{this}{:}p', \overline{x{:}p'}; \mathsf{p}$. So, by the substitution lemma (Lemma 2) and weakening of the linear, valid and locking contexts, we can conclude that $\Gamma; \Delta; \Psi; \mathsf{p} \vdash \quad [\overline{l/x}][l/\mathtt{this}]e : C_r.p_r \quad \dashv \Delta''_o; \mathsf{p}$.

The stack well-formedness property still holds. There were no active locks or active unpack before the step, and by the premise of P-METH-DECL, there are none after. Otherwise, $\kappa$ and $\Psi$ have not changed from before the step. The number of locks returned by $\mathsf{numLocks}$ has not changed, nor has $\kappa$. And finally, $u = u' = \mathsf{p}$, so $\mathsf{upProtect}$ is trivially true.

The single-thread guarantees still hold. The permission contexts and the heap have not changed.

The side-conditions hold:

- $C_r.p_r = C_r.p_r$
- True by the substitution and weakening lemmas. Substitution because $\Delta$ will not contain $\mathtt{this}$ or $\overline{x}$.
- True because $u = \mathsf{p}$ the same packing flag under which the method body was checked.
- $\Gamma' = \Gamma$.
- True, since active locks before and after are the same along with $\Psi$.
- Does not apply.

Finally, the heap invariant must hold because the static contexts are the same, and the run-time state has not changed.

**Case** (D-Spawn)**.**

For this case, $a = e_2$, so we have an additional proof burden for this newly spawned thread. Since the original expression must be well-typed, it must have been typed by rule P-SPAWN. And by the inversion of this rule we know that $\Gamma; \Delta; \Psi; \mathsf{p} \vdash \quad l : E, \overline{l : E} \quad \dashv \Delta_o; \mathsf{p}$, where $\downarrow^{\bullet} (l : E, \overline{l : E}) = l : C.p, \overline{l : C.p}$.

After the step, $e' = \mathtt{new}\ \mathsf{Object}()$ and $e_2 = l.m(\bar{l})$. Choose the linear contexts as follows: $\Delta_1 = \Delta_o$ and $\Delta_2 = l : C.p, \overline{l : C.p}$. We claim that $\Delta \vdash \quad \Delta_1$ and $\Delta \vdash \quad \Delta_2$ by examining the only typing rules that can be used to type labels $l$, P-TERM-I and P-TERM-II. These rules take

permissions from the incoming context $\Delta$ use them to type labels $l : C.p, \overline{l : C.p}$, and put the remainder in the outgoing context, $\Delta_o$. They must also be consistent with one another since they were effectively split from the same incoming context.

By P-CALL, $\Gamma; l{:}p, \overline{l{:}p}; \bullet; \mathsf{p} \vdash\ l.m(\bar{l}) : C_r.p_r\ \dashv l{:}p', \overline{l{:}p'}; \mathsf{p}$. We know that $\Gamma; l{:}p, \overline{l{:}p}; \bullet; \mathsf{p} \vdash\ l : C.p, \overline{l : C.p}\ \dashv \bullet; \mathsf{p}$ by the premise of the P-SPAWN rule. This is also why we know the permissions match the required pre-conditions for the method call.

By P-NEW, $\Gamma; \Delta_o; \Psi; \mathsf{p} \vdash\ \texttt{new}\ \mathsf{Object}() : C.\mathsf{unique@alive}\ \dashv \Delta_o; \mathsf{p}$. The Object constructor implicitly exists in our system and takes no arguments, so it trivially satisfies their permissions. The alive state implicitly exists for type Object, and has the trivially-true state invariant.

The heap invariant holds because our new contexts are the same with the exception of $\Delta_o$, and since $\Delta$ was only used to type labels $l$, $\Delta_o$ consists entirely of pass-through facts from $\Delta$ which were true before the step. Finally, $H = H', \rho = \rho'$ and $\kappa = \kappa'$.

**Case** (D-New)**.**

By the only applicable type rule, P-NEW, we know that the expression was typed as follows: $\Gamma; \Delta; \Psi; u \vdash\ \texttt{new}\ C(\bar{l}) : C.\mathsf{unique@}s\ \dashv \Delta_o; u$ and by inverting this rule we also know that $\Gamma; \Delta; \Psi; u \vdash\ \overline{l : C.p}\ \dashv \Delta_o; u$, where $\mathsf{init}(C) = \langle \overline{f : C}, s \rangle$.

By typing rule P-TERM-II, the resulting expression is well-typed, $\Gamma, o{:}C; \Delta_o, o{:}\mathsf{unique@}s; \Psi; u \vdash\ o : C.\mathsf{unique@}s\ \dashv \Delta_o; u$.

The stack well-formedness property continues to hold. Active locks and active unpack are well-defined for both $e$ and $e'$, although they are empty. $\Psi, \kappa$ along with activeLocks have stayed the same as they were before the step, so any relationships still hold. upProtect will continue to hold if $u = u' = \mathsf{up}(l; \mathsf{share}; S; \overline{f : p})$, since $\kappa = \kappa'$.

The single-threaded guarantees hold because this is a new object.

The side-conditions are true:

- $C.\mathsf{unique@}s = C.\mathsf{unique@}s$
- $\Delta_o = \Delta_o$
- $u_o = u_o$
- $\Gamma'$ is larger with the simple addition of $o{:}C$.
- Since $\mathsf{activeLocks}(e) = \mathsf{activeLocks}(e') = \emptyset$, and $\Psi = \Psi'$, this is true.
- $u' = u$, so this must hold

The heap invariant holds. The valid context is consistent with the heap because the rule D-NEW places $o \mapsto C(\overline{o})@s$ into the heap. Similarly, the new permission in $\Delta'$ which is $o{:}\mathsf{unique@}s$ is consistent. Additionally, $\mathsf{inv}_C$ is true for the fields of $o$ because $\Delta$ was used to prove the state invariant before the step, and it was consistent with the heap. The packing state has not changed so remains true, and there are no new share permissions in the linear context.

□

## B.3 Thread-Level Safety Lemmas

In this section there are a variety of lemmas that the single-threaded proof of safety depends upon.

**Lemma 2** (Substitution)**.** *Given a well-typed expression $e$ under a series of static contexts. Substituting $l$ for $t_1$ in $e$ uniformly, along with in the typing contexts will still result in a well-typed term. In other words, given $\Gamma; \Delta; \Psi; u \vdash \quad e : E \quad \dashv \Delta_o; u_o$, it is also the case that $[l/t_1]\Gamma; [l/t_1]\Delta; [l/t_1]\Psi; [l/t_1]u \vdash \quad [l/t_1]e : E \quad \dashv [l/t_1]\Delta_o; [l/t_1]u_o$.*

*Proof.* By induction on the cases of the typing judgment.

**Case** (P-Term-I)**.**

If $t_1 \neq b$ then the result is trivially true. The rest of the linear context and the entire packing context passes through, so that $\Delta_o = [l/t_1]\Delta, b{:}k''@S = [l/t_1](\Delta, b{:}k''@S)$ and $[l/t_1]u = [l/t_1]u_o$.

If $t_1 = b$, then we can use rule P-TERM-I to conclude the desired result, $\Gamma, l{:}C; \Delta, l{:}k@S; [l/t_1]\Psi; [l/t_1]u \vdash \quad l{:}C.k'@S' \quad \dashv \Delta, l{:}k''@S$.

**Case** (P-Term-II)**.**

This case is very similar to the previous case.

**Case** (P-Load)**.**

We need to consider two cases. First, if $t_r = t_1$, then P-LOAD will still hold, since the contexts are substituted uniformly. $\Gamma, l{:}C; [l/t_1]\Delta; [l/t_1]\Psi; [l/t_1]u \vdash \quad \overline{t_r.f_i^k} \quad \dashv [l/t_1]\Delta; [l/t_1]u'$ where $[l/t_1]u = \mathsf{up}(l; k_r; S_r\overline{f : p}, f_i{:}k_i@S_i)$ and $[l/t_1]u' = \mathsf{up}(l; k_r; S_r\overline{f : p}, f_i{:}k_i'@S_i)$.

Otherwise, if $t_r \neq t_1$, the result holds by nature of the fact that $\Delta$ is passed through to the output context so substitution will carry through.

**Case** (P-Assign)**.**

If $t_r = t_1$ then this case is very much like the previous case. Same holds for if $t_1 \neq t_r$ and $t_1 \neq t$.

However, if $t = t_1$, then by the induction hypothesis, $[l/t_1]\Gamma, t_r{:}C; [l/t_1]\Delta; [l/t_1]\Psi; [l/t_1]u \vdash \quad l : C_i.p \quad \dashv [l/t_1]\Delta_o; [l/t_1]u$.

**Case** (P-Unpack-Unique)**.**

If $t_1 = t$ then the result is true by this typing rule and the induction hypothesis, since $\Gamma, t_1{:}C; [t_1/l]\Delta; [t_1/l]\Psi; \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S)) \vdash \quad [t_1/l]e : E \quad \dashv [t_1/l]\Delta_o; \mathsf{up}(l; \mathsf{unique}; S; \mathsf{inv}_C(S))$.

Otherwise, the result still holds by the induction hypothesis, but $u$ is unchanged.

**Case** (P-InUnpack-Uniq)**.**

This case is very similar to the previous case.

**Case** (P-Unpack-Imm)**.**

This case is very similar to P-Unpack-Unique.

**Case** (P-InUnpack-Imm)**.**

This case is very similar to P-Unpack-Unique.

**Case** (P-Unpack-Share)**.**

This case is very similar to P-Unpack-Unique. The main difference here is that, if $t_1 = t_r$ the newly substituted lock context, $[l/t_1]\Psi$ will contain $l$, so the unpacking will remain protected by a lock.

**Case** (P-InUnpack-Share)**.**

This case is very similar to P-Unpack-Share.

**Case** (P-Sync)**.**

If $t_1 = t$, then the substituted expression would be, $\texttt{synchronized}\ (l)\ [l/t_1]e$. By the P-SYNC typing rule, and the induction hypothesis, the entire expression would be well-typed.

**Case** (P-InSync)**.**

If $t_1 = l$, then the resulting expression will be $\texttt{insync}(l')\ [l'/t_1]e$. $l'$ will also be substituted for $t_1$ in $\Psi$, so the rule will still apply, and by the induction hypothesis, the output contexts are correct.

**Case** (P-Let)**.**

Because of alpha conversion, we can implicitly assume that $x \neq t_1$. After that, we can simply rely on the induction hypothesis. It tells us that $[l/t_1]\Gamma; [l/t_1]\Delta; [l/t_1]\Psi; [l/t_1]u \vdash \ e_1 : C_1.p_1 \ \dashv [l/t_1]\Delta_1; [l/t_1]u_1$. By the induction hypothesis again, $[l/t_1]\Gamma, x{:}C_1; \downarrow^{\Psi}([l/t_1]\Delta), x{:}\downarrow(p_1); [l/t_1]\Psi; [l/t_1]u \vdash \ e_2 : E \ \dashv [l/t_1]\Delta_o, x{:}p_x; u_o$. By the P-LET rule, the entire expression is well-typed.

**Case** (P-Call)**.**

If $t = t_1$ or $t_i = t_1$ for any of the method arguments, by the induction hypothesis, the newly substituted term would still be well-typed, and the resulting output context would equal $[l/t_1]\Delta_o$.

Assume for example that $t = t_1$, but this would apply for any of the arguments. From the P-CALL rule itself, the resulting output linear context for the entire expression would be $[l/t_1]\Delta_o, l{:}p', \overline{t : p'}$, which satisfies the result.

**Case** (P-Spawn)**.**

This case is very similar to P-Call.

**Case** (P-New)**.**

This case is very similar to P-Call.

□

**Lemma 3** (Valid Weakening)**.** *If an expression is well-typed under a valid context* $\Gamma$*, then it is well-typed under a larger context* $\Gamma'$*. In other words, given* $\Gamma; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta_o; u_o$ *and a context* $\Gamma'$ *such that* $\Gamma \leq \Gamma'$*, then* $\Gamma'; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta_o; u_o$*.*

*Proof.* By induction on the cases of the typing derivation. By looking at all the typing rules, we can see that, while some require a certain label $b$ to be present in the valid context, none of the rules limit what can be in the "rest" of the context, and therefore will not become invalid when entries are added. □

**Lemma 4** (Linear Weakening)**.** *If an expression is well-typed under a linear context $\Delta$, then it is well-typed under a larger context $\Delta'$. In other words, given $\Gamma; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta_o; u_o$ and a context $\Delta'$ such that $\Delta \leq \Delta'$, then $\Gamma; \Delta'; \Psi; u \vdash \ e : E \ \dashv \Delta'_o; u_o$ and $\Delta_o \leq \Delta'_o$.*

*Proof.* By induction on the cases of the typing derivation. First, consider the cases P-TERM-I and P-TERM-II. If $\Delta$ is increased by adding more permissions, then those permissions will simply be passed through to $\Delta_o$. If the permission to $b$ is made stronger, then by definition of stronger, rule P-TERM-I can be used to split the new stronger permission into the old permission. The expression keeps the same type, and the remaining permission is added to $\Delta_o$ making it larger.

Four of the cases, P-ASSIGN, P-CALL, P-SPAWN and P-NEW make direct use of the typing rules P-TERM-I and P-TERM-II, and use the resulting output context as the output context for the entire expression. These cases are true then by appealing to the induction hypothesis.

The unpack, inunpack, synchronized and insync cases are all extremely similar. They directly pass the linear context into their subexpression $e$, and use the resulting output context $\Delta_o$ as the output context for the entire expression. By the induction hypothesis, then, these cases are true.

Finally, for case P-LET, we again appeal to the induction hypothesis. First, to show that expression $e_1$ remains well-typed and gives us a new, larger context $\Delta'_1$. We appeal to the induction hypothesis again to prove that $e_2$ is well-typed, and produces a new, larger output context $\Delta'_o$. □

**Lemma 5** (Lock Weakening)**.** *For a well-typed expression, $\Gamma; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta'; u'$, we can add $l$ from the context of held locks, and the expression remains identically typed I.e., it is the case that $\Gamma; \Delta; \Psi, l; u \vdash \ e : E \ \dashv \Delta'; u'$.*

*Proof.* By induction on the cases of the typing derivation. Like with valid weakening, we can look at each of the typing rules. While some typing rules do require a particular label to be present in the lock context, none of the typing rules restrict what *else* can be in the lock context, and therefore no well-typed expression will become mis-typed under an enlarged locking context. □

**Lemma 6** (Unpacking Weakening)**.** *If an expression is well-typed under an packing flag $u$, then it is well-typed under a larger packing flag $u'$. In other words, given $\Gamma; \Delta; \Psi; u \vdash \ e : E \ \dashv \Delta_o; u_o$ and a packing flag $u'$ such that $u \leq u'$, then $\Gamma; \Delta; \Psi; u' \vdash \ e : E \ \dashv \Delta_o; u'_o$ and $u_o \leq u'_o$.*

*Proof.* By induction on the cases of the typing derivation. Most cases either ignore $u$ or appeal to the induction hypothesis directly.

For P-LOAD, we can consider whether $u$ was made larger by adding new fields or by adding stronger permissions. If the former, the expression remains well-typed because it simply ignores all other field permissions besides $f_i$. If the latter, then a stronger field permission is essentially defined by the ability to split it into the original permission. In this case, it can be split, and the remaining permission goes into the output packing flag, making it larger.

For P-ASSIGN, the reasoning is quite similar. In this case, if $k_i$ is increased to a stronger permission, $k_i \vdash \ k$ will still hold for the original $k$, since that is what it means to be a stronger permission.

All of the unpack and inunpack cases are interesting because the output packing context is required to prove some state invariant, $\mathsf{inv}_C(S)$. These cases appeal to the induction hypothesis to show that, for a larger input packing flag $u$, the output packing flag $u_o$ will be larger than the original output packing flag. And again, it is essentially the definition of $\leq$ that if the permissions associated with the new output packing flag are larger, they will still be able to prove $\mathsf{inv}_C(S)$. □

**Lemma 7** (Lock Decrementing)**.** *In a single step an expression with an active lock can add or remove at most one lock from the number of active locks in an expression. In other words, If* $\mathsf{numLocks}(e, \rho, o) = i$ *and* $e$ *can take a step,* $e \to e'$*, then* $\mathsf{numLocks}(e', \rho', o) = j$ *where* $j = i \vee j = i - 1 \vee j = i + 1$.

*Proof.* By induction on the single-threaded transition relation. By examining all of the transitions, we can see that each one adds or removes at most one `insync` from the expression at a time. Additionally, no rule that allows a subexpression to take a step allows more than one subexpression to take a step. □

**Lemma 8** (Lock Strengthening)**.** *For a well-typed value,* $\Gamma; \Delta; \Psi, l; u \vdash \quad o : E \quad \dashv \Delta'; u'$*, we can remove* $l$ *from the context of held locks, and the value remains identically typed I.e., it is the case that* $\Gamma; \Delta; \Psi; u \vdash \quad o : E \quad \dashv \Delta'; u'$.

*Proof.* By induction on the cases of the typing derivation judgment. □

**Lemma 9.** *No precise* share *permission for a field can ever be in the packing flag* $u$ *in a well-typed program.*

*Proof.* By induction on the cases of the transition relation. The interesting cases are the unpacking "enter" cases and the D-ASSIGN. For the enter cases, the new unpacking after the step is completed we will choose the new packing flag $u'$ to be $\mathsf{up}(l; k; S; \mathsf{inv}_C(S))$ or $\mathsf{up}(l; \mathsf{immutable}; s; \mathsf{purify}(\mathsf{inv}_C(s))$ depending on whether or not an immutable unpack is being performed. The predicate $\mathsf{inv}_C$ uses the declared state invariants for each state or the empty list if $S = ?$. The empty list contains no precise permissions. For a program to be considered valid, according to P-SINV, no state invariant can have a precise permission.

For the assignment case, the new packing flag $u'$ after the step is $\mathsf{up}(l; k; S; \overline{f : p}, f_i :\downarrow (p))$. We know that no share permissions in $\overline{f : p}$ are precise because this property held before the step. And, because $p$ is being downgraded, it will also not contain any precise share permissions.

□

# Bibliography


[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *The 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37. ACM Press, 2006. 2, 8.2.1, 8.4

[2] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *In ECOOP '04: Proceedings of the European Conference on Object-Oriented Programming*, pages 1–25. Springer-Verlag, 2004. 5.6

[3] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003. 8.4

[4] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022. ACM Press, 2009. 1.6

[5] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, New York, NY, USA, 2005. ACM Press. 2.2.2, 2.5

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009. ISSN 0001-0782. 1.1

[7] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001. 2.1

[8] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology Special Issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6):27–56, June 2004. 3.2.2, 3.5.1

[9] Nels E. Beckman and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate: Technical companion. Technical Report CMU-ISR-08-126, Carnegie Mellon University, 2008.

http://reports-archive.adm.cs.cmu.edu/anon/isr2008/CMU-ISR-08-126.pdf. 3.3.5, 4.1, 4.2.2

[10] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *The 2008 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2008. 2.1, 3, 3.3.5, 8.3.1

[11] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. Reducing stm overhead with access permissions. In *IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 1–10. ACM Press, 2009. 8

[12] Josh Berdine, Cristiano Calcagno, and Peter W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005. 3.5.1

[13] Kevin Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, April 2009. 3.2.2, 3.3.4, 3.4, 5.3.2, 6.1.1, 6.1.3, 6.4, 7.4, 7.4

[14] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 217–226, September 2005. 2.1, 3.2.2

[15] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *The 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320. ACM Press, 2007. 1.2, 1.3.2, 2.1, 2.4.2, 2.4.4, 3, 3.1, 3.2.1, 3.2.2, 3.3, 3.3.1, 3.3.2, 3.3.3, 3.3.3, 3.3.4, 3.3.4, 3.3.4, 3.5.1, 3.7, 4.1, 5.1, 5.3.5, 5.3.6, 5.6, 9.4

[16] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*, pages 195–0219, July 2009. 2.4.3, 5.1, 6.5

[17] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007. 5.6

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190. ACM Press, 2006. 6.5

[19] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999. 8.4

[20] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional

memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006. 3.3.5

[21] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47. ACM Press, 2008. 6.1.1

[22] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005. 3.5.1, 3.5.1, 5.6

[23] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004. 5.6

[24] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002. 1.2, 3.1, 3.5.2

[25] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer. 1.3.2, 3.1, 3.3, 3.3.2, 3.5.1, 5.6

[26] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375: 227–270, April 2007. 3.5.1

[27] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *The 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999. 8.4

[28] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977. 3.4

[29] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982. 7.1

[30] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5):59–69, 2001. 1.2, 2.1

[31] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '04: European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004. 1.2, 3.2.1, 3.2.2, 5.3.5

[32] Peter J. Denning and Jack B. Dennis. The resurgence of parallelism. *Commun. ACM*, 53 (6):30–32, 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1743546.1743560. 9.3.2

[33] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006. 8.2.3

[34] Werner Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, December 2009. 7.6

[35] Werner Dietl and Peter Mller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–43, 2005. 3.5.1, 6.4, 9.2

[36] Werner Dietl, Sophia Drossopoulou, and Peter Mller. Generic universe types. In *In ECOOP '07: Proceedings of the European Conference on Object-Oriented Programming*, pages 28–53. Springer, 2007. 5.6

[37] Dino Distefano and Matthew J. Parkinson J. jstar: towards practical verification for java. In *The 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM Press. 3.5.1

[38] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 281–292. ACM Press, 2004. 2.2.1, 2.3.2

[39] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252. ACM Press, 2003. 3.1, 3.5.2

[40] Manuel Fahndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 337–350. ACM Press, 2007. 2.1, 6.1.3

[41] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 762–763. ACM Press, 2006. 2.1

[42] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17 (2):1–34, 2008. 2.1

[43] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, 2000. 1.2, 3.2.3, 3.6

[44] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003. 3.5.2

[45] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006. 2.2.1, 2.3.2

[46] Stephen Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003. 3.5.1

[47] Ronald Garcia, Roger Wolff, Éric Tanter, and Jonathan Aldri ch. Featherweight typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University, July 2010. 4.2.1, 4.2.2

[48] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987. 3.2.1, 5.6

[49] Madhu Gopinathan and Sriram K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 245–260. ACM Press, 2008. 2.1

[50] Aaron Greenhouse and William L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 453–463. ACM Press, 2002. 3.2.3, 3.5.2, 3.6

[51] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25. ACM Press, 2003. 3.5.2

[52] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. 3.2.2

[53] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct 2003. 3.3.5, 8.1

[54] Tim Harris and Simon Peyton Jones. Transactional memory with data invariants. In *TRANSACT '06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006. 3.5.1

[55] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006. 8.4

[56] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13. ACM Press, 2004. 3.5.2

[57] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993. 3.3.5, 8.1

[58] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *The twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003. 3.3.5, 8.2.1

[59] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *The 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262. ACM Press, 2006. 3.3.5, 8.3.1

[60] Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 344–368. Springer-Verlag, 2009. 2.2.2, 2.5, 9.2, 9.3.1

[61] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *TRANSACT '06: First ACM SIGPLAN Workshop on Languages, Compilers,*

*and Hardware Support for Transactional Computing*, 2006. 3.5.2

[62] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *The 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM Press, 2006. 2, 8.2.1

[63] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. 3.3, 5.3.1

[64] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society. 3.1, 3.2.3, 3.5.1, 3.6

[65] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2009. Springer-Verlag. 2.1, 2.2.4

[66] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983. 3.1, 3.5.1

[67] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 288–296, Sept. 2008. 2.1, 2.2.2, 6.5

[68] Duri Kim. An empirical study on the frequency and classification of object protocols in java. Master's thesis, Korea Advanced Institute of Science and Technology, 2010. 2, 2.3.2

[69] Yoon Phil Kim. Permission-based optimization for effecient software transactional memory. Master's thesis, Carnegie Mellon University, 2008. 8

[70] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176. USENIX Association, 2006. 7.6

[71] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004. 2.1

[72] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975. 3.5.1, 3.5.2

[73] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. *SIGPLAN Not.*, 44(6):75–86, 2009. ISSN 0362-1340. 7.6

[74] Thomas Minka. Expectation propagation for approximate bayesian inference. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 362–369. Morgan Kaufmann, 2001. 7.3

[75] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62. ACM Press, 2008.

4.2.5, 4.2.6

[76] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 347–366. ACM Press, 2008. 2.4.2, 6.5

[77] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375 (1-3):271–307, 2007. 3.5.1

[78] Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. 3.1, 3.5.1

[79] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge. 3.5.1

[80] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 311–324. ACM Press, 2006. 5.6

[81] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331. ACM Press, 2006. 3.1, 3.5.2

[82] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 53–65. ACM Press, 2009. 2.1, 6.1.3

[83] Qualitas Research Group. Qualitas corpus version 20090202r, `http://www.cs.auckland.ac.nz/~ewan/corpus`. University of Auckland, February 2009. 2.2.3, 2.2.4

[84] Edwin Rodriguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP '05: Object-Oriented Programming 19th European Conference*, pages 551–576, 2005. 3.1, 3.5.1

[85] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94. ACM Press, 2005. 1.2, 3.5.2

[86] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997. 3.5.2

[87] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Notices*, 42(6):78–88, 2007. 8.4

[88] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceed-*

*ings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 224–244, London, UK, 2000. Springer-Verlag. 3.5.2

[89] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *In Proceedings of Onward! Conference*, October 2009. 1.6, 9.3.2

[90] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. 1.2, 2.1, 3.1, 3.2.1

[91] Jeffrey Stylos. *Making APIs More Usable with Improved API Designs, Documentation and Tools*. PhD thesis, Carnegie Mellon University, May 2009. 9.3.2

[92] Tachio Terauchi. Checking race freedom via linear programming. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–10. ACM Press, 2008. 4.2.6, 7.6

[93] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–345. ACM, 2006. 3.1, 3.5.1

[94] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359, 1990. 3.2.2

[95] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, November 2009. 2.5

[96] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. *SIGPLAN Not.*, 39(10):419–431, 2004. 2.5, 6.5, 9.2

[97] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 218–228. ACM Press, 2002. 2.2.2, 2.5

[98] Hirotoshi Yasuoka and Tachio Terauchi. Polymorphic fractional capabilities. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 36–51, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03236-3. doi: http://dx.doi.org/10.1007/978-3-642-03237-0_5. 5.6

[99] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234. ACM Press, 2005. 3.5.2

[100] Yang Zhao. *Checking Interference with Fractional Permissions*. PhD thesis, University of Wisconsin-Milwaukee, August 2007. 3.3

[101] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on Object-*

*Oriented Programming*, pages 318–343. Springer-Verlag, 2009. 2.2.2, 2.5, 9.2, 9.3.1