SafeJava: A Unified Type System for Safe Programming

by

Chandrasekhar Boyapati

B.Tech. Indian Institute of Technology, Madras (1996)
S.M. Massachusetts Institute of Technology (1998)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author ....
Chandrasekhar Boyapati
Department of Electrical Engineering and Computer Science
February 2, 2004

Certified by ....
Martin C. Rinard
Associate Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

SafeJava: A Unified Type System for Safe Programming
by
Chandrasekhar Boyapati

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Making software reliable is one of the most important technological challenges facing our society today. This thesis presents a new type system that addresses this problem by statically preventing several important classes of programming errors. If a program type checks, we guarantee at compile time that the program does not contain any of those errors.

We designed our type system in the context of a Java-like object-oriented language; we call the resulting system *SafeJava*. The SafeJava type system offers significant software engineering benefits. Specifically, it provides a statically enforceable way of specifying object encapsulation and enables local reasoning about program correctness; it combines effects clauses with encapsulation to enable modular checking of methods in the presence of subtyping; it statically prevents data races and deadlocks in multithreaded programs, which are known to be some of the most difficult programming errors to detect, reproduce, and eliminate; it enables software upgrades in persistent object stores to be defined modularly and implemented efficiently; it statically ensures memory safety in programs that manage their own memory using regions; and it also statically ensures that real-time threads in real-time programs are not interrupted for unbounded amounts of time because of garbage collection pauses. Moreover, SafeJava provides all the above benefits in a common unified type system framework, indicating that seemingly different problems such as encapsulation, synchronization issues, software upgrades, and memory management have much in common.

We have implemented several Java programs in SafeJava. Our experience shows that SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. In addition, the type declarations in SafeJava programs serve as documentation that lives with the code, and is checked throughout the evolution of code. The SafeJava type system thus has significant software engineering benefits and it offers a promising approach for improving software reliability.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor, Electrical Engineering and Computer Science

Acknowledgments

I have benefited greatly from the wisdom and kindness of many people.

First, I must thank my advisor Martin Rinard for his guidance and support. His energy, excitement, and enthusiasm have been very contagious. He not only helped me with the technical aspects of my research but he also taught me a great deal about the broader social context in which the research takes place, and helped prepare me for a career in research.

Barbara Liskov has been a continuous source of counsel and support during the course of my graduate study. I spent my first four years at MIT in her group. I particularly thank her for the freedom she gave me to explore my ideas during this time. While this freedom took me down some garden paths, it taught me a lot about the research process and made me a better researcher and a better individual.

I am grateful to several colleagues who collaborated with me and gave me useful feedback on my research. Robert Lee worked with me on a prototype implementation of SafeJava. Alexandru Salcianu collaborated with me on extending SafeJava to do safe region-based memory management. William Beebee implemented the Real-Time Java platform. Liuba Shrira worked with us on the software upgrades project. Chuang-Hue Moh, Steven Richman, and Dorothy Curtis implemented the upgrades infrastructure in Thor. Viktor Kuncak and Alexandru Salcianu gave several insightful suggestions throughout the course of this research. Butler Lampson provided useful comments as a thesis reader.

Outside the work presented in this thesis, Darko Marinov and Sarfraz Khurshid collaborated with me on Korat, a framework for automated testing of software. Gregor Kiczales was my mentor at Xerox Parc where I did an internship in the Aspect Oriented Programming group. Erik Ruf was my mentor at Microsoft Research where I did an internship in the Marmot Compiler group.

My friends have made my stay at MIT an enjoyable one. I must thank members of my current group including Viktor, Alex, Maria, Sarfraz, Darko, Patrick, Karen, Brian, Scott, and Wes; members of my previous group including Atul, Jason, Andru, Kavita, Miguel, Chon Chon, Yan, Rodrigo, and Zheng; other members of the fifth floor including Dina, Rahul, Bodhi, Kyle, Jinyang, and Xiaowei; and members of the theory group including Eric, Danny, Yevgeniy, Matthius, and Venkat. They not only made the lab a pleasant and relaxed environment to work in, but have been wonderful companions during the course of my graduate study. Q, Laura, Kyle, and Yan were, to a large extent, responsible for ensuring that I had a life outside the lab. Prasad, Karasi, Hiran, and Discu have been terrific house mates. Mixie, Janaki, Ninja, Kanta, Lallu, Vishnu, and JJ have been good friends since IIT times; and Shamibrata, Raja, and Aveek since high school.

Finally, I thank my family for their continued love and support. My parents Nagarathnam and Gangulaiah have always encouraged me to aim high and to pursue my dreams. They have done whatever they could to ensure that I had the best education possible. I owe them much gratitude. My brother Shyam has been one of my closest friends and a continuous source of support. Last but not the least, I thank my newly-wed wife Lahari for her unconditional love and for the promise of wonderful times ahead.

Contents

List of Figures

Chapter 1

Introduction

The motivation behind this thesis is the need for reliable software. Software is rapidly becoming the foundation for our entire civil infrastructure. All activities including transportation, telecommunications, energy, medicine, and banking rely on the correct working of software systems. As software becomes more pervasive in our infrastructure, failures of software can cause more and more damage. Hence the increasing need for reliable software. Software reliability also has a significant impact on our economy. Studies estimate that bugs in software cost businesses worldwide about $175 billion in the year 2001 [124]. Making software reliable is one of the most important problems facing computer science today. Making software reliable is also one of the most challenging problems, primarily because of the inherent complexity of large software systems.

This thesis presents a new type system that improves software reliability by preventing several classes of common but potentially serious programming errors. If a program type checks, the system guarantees at compile time that the program does not contain any of those errors. We designed our type system in the context of a Java-like object-oriented language; we call the resulting system *SafeJava*.

The SafeJava type system offers significant software engineering benefits. Specifically, SafeJava provides a statically enforceable way of specifying object encapsulation and enables local reasoning about program correctness; it combines effects clauses with encapsulation to enable modular checking of methods in the presence of subtyping; it statically prevents data races and deadlocks in multithreaded programs, which are some of the most difficult programming errors to detect, reproduce, and eliminate; it enables software upgrades in persistent object stores to be defined modularly and implemented efficiently; it statically ensures memory safety in programs that manage their own memory using regions; and it also statically ensures that real-time threads in real-time programs are not interrupted for unbounded amounts of time. Moreover, SafeJava provides all these benefits in a unified type system framework, indicating that seemingly different issues such as encapsulation, synchronization, upgrades, and memory management have much in common.

We have implemented several Java programs in SafeJava. Our experience shows that SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. In addition, the type declarations in SafeJava programs serve as documentation that lives with the code, and is checked throughout the evolution of code. The SafeJava type system thus offers a promising approach for improving software reliability.

The rest of this introductory chapter is organized as follows. Sections 1.1 to 1.5 elaborate on the various benefits offered by the SafeJava type system. Section 1.6 highlights the contributions of this thesis. Section 1.7 describes the structure of the subsequent chapters.

1.1 Enforcing Object Encapsulation

The ability to reason locally about program correctness is crucial when dealing with large programs. Local reasoning allows correctness to be dealt with one module at a time. The standard approach is to provide each module with a specification describing its expected behavior. The goal is to prove that each module satisfies its specification, using only the specifications but not code of other modules. This way the complexity of the proof effort (formal or informal) can be kept under control.

This local reasoning approach is sound if separate verification of individual modules suffices to ensure the correctness of the composite program [98, 56]. The key to sound local reasoning in object-oriented languages is object encapsulation. Consider, for example, a Stack object s that is implemented using a linked list. Local reasoning about the correctness of the Stack implementation is possible if objects outside s do not directly access the list nodes, i.e., the list nodes are *encapsulated* within the s.

SafeJava uses a variant of ownership types for specifying and statically enforcing object encapsulation. In SafeJava, a program can declare that s *owns* all the list nodes. The type system then statically ensures that the list nodes are encapsulated within s.

A type system that strictly enforces object encapsulation, however, is too constraining [113]: it does not allow efficient implementation of important constructs like iterators [104, 71]. Consider, for example, an iterator over the above-mentioned Stack object s. If the iterator is encapsulated within s, it cannot be used outside s. If the iterator is *not* encapsulated within s, it cannot directly access the list nodes in s, and hence cannot run efficiently.

Previous ownership type systems were either too constraining to support constructs like iterators [43, 42], or too permissive to support local reasoning [41]; for example they allowed objects outside the above-mentioned Stack object s to temporarily get direct access to the list nodes.

This thesis argues that the right way to solve the problem is to provide special access privileges to objects belonging to classes in the same module; we show how to do this for inner classes [107, 89]. SafeJava allows inner class objects to have privileged access to the representations of the corresponding outer class objects. This principled violation of encapsulation allows programmers to express constructs like iterators using inner classes, yet supports local reasoning about the correctness of the classes. SafeJava supports local reasoning because a class and its inner classes can be reasoned about together as a module.

SafeJava also allows programmers to use unique pointers [108]. Like ownership types, unique pointers are useful to constrain object aliasing. Because SafeJava combines ownership types and unique pointers in a common framework, it supports constructs that neither ownership types nor unique pointers alone can support, while enforcing object encapsulation.

1.2 Checking Side Effects of Methods Modularly

SafeJava combines object encapsulation with effects clauses [106] which are useful for specifying assumptions that must hold at method boundaries and enable modular checking of programs.

Effects clauses have been incorporated into many program formalisms, specification languages, and program checkers. Examples include Morgan's specification statement [110], Z [126], Larch [82], JML [96], and ESC [57, 69]. SafeJava also uses effects clauses to statically verify various program properties.

In a language that does not have a notion of object encapsulation, it is often difficult to specify the side effects of a method precisely without exposing the representation of its object. Consider, for example, the modifies clause [103] on the push method of a Stack object that is implemented using a linked list. The modifies clause specifies the names of all the objects that the corresponding method may modify. Suppose we want to check the modifies clause automatically with a program analysis tool. Since the push method may modify the list nodes, the modifies clause must specify the names of the list nodes, thus exposing the representation of the Stack object.

The problem gets worse if there are several Stack implementations. For example, in an object-oriented language like Java, one can have an abstract Stack class and different subclasses of Stack that use different representations. The subclasses add new fields to Stack, and the push method in a subclass might modify the objects pointed to by the new fields. In this case it is impossible to specify the modifies clause of the push method of Stack, because the new fields added by subclasses are not visible in the scope of the abstract Stack class.

To solve this problem, one must use some abstraction mechanism with which one can refer to the objects that are not in scope without directly mentioning their names. Data groups [97, 99] are one such abstraction mechanism. SafeJava provides an alternate solution. SafeJava allows effects clauses to use the name of an object o to denote all the objects (reflexively and transitively) encapsulated within o. The push method of Stack can declare that it modifies the corresponding Stack object. If a subclass adds a new field f, it can declare that the object pointed to by f is encapsulated within the Stack object. The push method in the subclass is then allowed to modify the object pointed to by f.

SafeJava thus allows programs to specify the side effects of a method precisely in the presence of subtyping without representation exposure.

1.3 Preventing Data Races and Deadlocks

Multithreaded programming is becoming a mainstream programming practice. But multithreaded programming is difficult and error prone. Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. A deadlock occurs when there is a set of threads such that every thread in the set is waiting on a lock held by another thread in the set. Synchronization errors in multithreaded programs are timing-dependent, non-deterministic bugs, and are among the most difficult programming errors to detect, reproduce, and eliminate.

SafeJava provides a new static type system for multithreaded programs; well-typed programs in SafeJava are guaranteed to be free of data races and deadlocks. The basic idea is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. SafeJava allows programmers to specify this locking discipline in their programs in the form of type declarations. SafeJava statically verifies that a program is consistent with its type declarations.

To prevent data races, programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. The protection mechanism of an object can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either 1) the object is immutable, 2) the object is accessible to a single thread, or 3) there is a unique pointer to the object. Unique pointers are useful to support object migration between threads. The SafeJava type checker statically verifies that a program uses objects only in accordance with their declared protection mechanisms.

The SafeJava type system is significantly more expressive than previously proposed type systems for preventing data races [68, 11]. In particular, SafeJava lets programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. SafeJava does this by introducing a way of parameterizing classes that lets programmers defer the protection mechanism decision from the time when a class is defined to the times when objects of that class are created.

To prevent deadlocks, programmers partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The SafeJava type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order of lock levels. SafeJava allows programmers to write code that is polymorphic in lock levels. Programmers can specify a partial order among formal lock level parameters using `where` clauses [50, 112].

SafeJava also allows programmers to use recursive tree-based data structures to further order the locks within a given lock level. For example, programmers can specify that nodes in a tree must be locked in the *tree order*. SafeJava allows mutations to the data structure that change the partial order at runtime. The SafeJava type checker uses an intraprocedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime.

The SafeJava type system combines object encapsulation with safe multithreading. Object encapsulation is useful for safe multithreading because the same lock that protects an object can also protect the objects encapsulated within that object.

1.4 Enabling Safe Upgrades in Persistent Object Stores

Persistent object stores provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. Providing a satisfactory way of upgrading objects in a persistent object store has been a long-standing challenge. A natural way to define upgrades is for programmers to provide a *transform function* [138] for each class whose objects need to be upgraded. A transform function initializes the new form of an object using its current state. The system carries out the upgrade by using the transform functions to transform all objects whose classes are being replaced.

This way of handling upgrades introduces two problems:

1. The system must provide good semantics that let programmers reason about their transform functions locally, thus making it easy to design correct upgrades.

2. The system must run upgrades efficiently, both in space and time.

This thesis provides solutions to both problems.

The thesis first introduces a set of *upgrade modularity conditions* that constrain the behavior of an upgrade system. Any upgrade system that satisfies the conditions guarantees that when a transform function runs, it only encounters object interfaces and invariants that existed when its upgrade was defined. The conditions thus allow transform functions to be defined *modularly*: a transform function can be considered an extra method of the class being replaced, and can be reasoned about like the rest of the class. This is a natural assumption that programmers would implicitly make in any upgrade system—our conditions provide a grounding for this assumption. This way an upgrade system provides good semantics to programmers who design upgrades.

The thesis then shows how upgrades implemented in SafeJava can execute efficiently, while satisfying the upgrade modularity conditions. Previous approaches do not provide a satisfactory solution to this problem. An upgrade system could satisfy the conditions by keeping old versions of all objects, since old versions preserve old interfaces and old object states. However versions are expensive, and to be practical, an upgrade system must avoid them most of the time. Some earlier systems [116, 13, 100] avoid versions by severely limiting the expressive power of upgrades (e.g., transform functions are not allowed to make method calls); others [8, 114] limit the number of versions using a stop-the-world approach that shuts down the system for upgrade and discards the versions when the upgrade is complete; yet others [138] do not satisfy the upgrade modularity conditions that enable programmers to reason about their upgrades locally.

Our approach exploits the fact most transform functions are *well behaved*: they access only the object being transformed and its encapsulated objects. SafeJava statically checks if transform functions are well behaved. If they are, the runtime system provides an efficient way to enforce the upgrade modularity conditions without maintaining versions. If they aren't, we provide an additional mechanism, *triggers*, which can be used to control the order of transform functions to satisfy the conditions. If even triggers are insufficient, we use versions but only in cases where they are needed.

1.5 Enabling Safe Region-Based Memory Management

The Real-Time Specification for Java (RTSJ) [18] provides a framework for building real-time systems. The RTSJ allows a program to create real-time threads with hard real-time constraints. These real-time threads cannot use the garbage-collected heap because they cannot afford to be interrupted for unbounded amounts of time by the garbage collector. Instead, the RTSJ allows these threads to use objects allocated in immortal memory (which is never garbage collected) or in regions [128]. Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access heap references.

SafeJava introduces a new static type system for writing real-time programs in Java. SafeJava guarantees that the RTSJ runtime checks will never fail for well-typed programs.

The SafeJava type system makes several important technical contributions over previous type systems for region-based memory management. For object-oriented programs, it combines region types [39, 48, 80, 128] and ownership types [23, 24, 26, 41, 43] in a unified type system framework. Region types statically ensure that programs never follow dangling references. Ownership types statically enforce object encapsulation and enable modular reasoning about program correctness in object-oriented programs. Consider, for example, a Stack object s that is implemented using a Vector object v. To reason locally about the correctness of the Stack implementation, a programmer must know that v is not directly accessed by objects outside s. With ownership types, one can declare that s *owns* v. The type system then statically ensures that v is encapsulated within s.

In an object-oriented language that has only region types (e.g., [39]), the types of s and v would declare that they are allocated in some region r. In an object-oriented language that only has ownership types, the type of v would declare that it is owned by s. SafeJava provides a simple unified mechanism to declare *both* properties. The type of s can declare that it is allocated in r and the type of v can declare that it is owned by s. SafeJava then statically ensures that both objects are allocated in r, that there are no pointers to v and s after r is deleted, and that v is encapsulated within s. SafeJava thus combines the benefits of region types and ownership types.

SafeJava extends region types to multithreaded programs by allowing explicit memory management for objects shared between threads. It allows threads to communicate through objects in *shared regions* in addition to the heap. A shared region is deleted when all threads exit the region. However, programs in a system with only shared regions (e.g., [79]) will have memory leaks if two long-lived threads communicate by creating objects in a shared region. This is because the objects will not be deleted until both threads exit the shared region. To solve this problem, SafeJava introduces *subregions* within a shared region. A subregion can be deleted more frequently, for example, after each loop iteration in the long-lived threads.

SafeJava introduces *typed portal fields* in subregions to serve as a starting point for inter-thread communication, and user-defined *region kinds* to support subregions and portals.

SafeJava extends region types to real-time programs by statically ensuring that real-time threads do not interfere with the garbage collector. SafeJava augments region kind declarations with *region policy* declarations. It supports two policies for creating regions as in the RTSJ. A region can be an LT (Linear Time) region, or a VT (Variable Time) region. Memory for an LT region is preallocated at region creation time, so that allocating an object in an LT region only takes time proportional to the size of the object (because all the bytes have to be zeroed). Memory for a VT region is allocated on demand, so that allocating an object in a VT region takes variable time. SafeJava checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions.

SafeJava also prevents an RTSJ *priority inversion* problem. In the RTSJ, any thread entering a region waits if there are threads exiting the region. If a regular thread exiting a region is suspended by the garbage collector, then a real-time thread entering the region might have to wait for an unbounded amount of time. SafeJava statically ensures that this priority inversion problem cannot happen.

1.6 Contributions

This thesis presents SafeJava, a new type system that improves software reliability by preventing several classes of common but potentially serious programming errors. The thesis is based on several papers that we previously published [22, 23, 24, 25, 26, 27], and it makes the following contributions:

- **Enforcing Object Encapsulation:** Object encapsulation is key to sound local reasoning in object-oriented languages. SafeJava is the first ownership type system that can express constructs like iterators while also supporting local reasoning. SafeJava is also the first system that combines object encapsulation with effects clauses, unique pointers, and immutable objects.

- **Combining Object Encapsulation With Effects Clauses:** Effects clauses are useful for specifying assumptions that must hold at method boundaries and enable modular checking of programs. SafeJava combines object encapsulation with effects clauses to allow programs to precisely specify the side effects of a method in the presence of subtyping and without representation exposure. SafeJava allows effects clauses to use the name of an object to denote all the objects encapsulated within that object. SafeJava first combined object encapsulation with effects clauses in [26] to prevent data races in multithreaded programs. SafeJava also uses effects clauses to statically verify several other program properties.

- **Combining Object Encapsulation With Unique Pointers:** SafeJava combines ownership types with unique pointers to express constructs that neither ownership types nor unique pointers alone can express, while enforcing object encapsulation. We first combined ownership types with unique pointers in [26] to support ownership transfer. Recent work [44] proposes a more flexible approach that allows a program to specify a unique *external* pointer to an object; there can be other pointers to that object from objects encapsulated within it. We subsequently adopted this approach.

- **Combining Object Encapsulation With Immutability:** Immutable objects have many advantages. Unlike mutable objects, they can be shared across multiple aliases without complicating the task of understanding and reasoning about correctness of programs. SafeJava is the first system that extends the notion of immutability to object encapsulation. SafeJava statically verifies that if an object is declared to be immutable, then the program does not modify that object or objects encapsulated within that object. SafeJava first combined ownership types with immutable objects in [26] to allow multiple threads to access an immutable object (and its encapsulated objects) without synchronization and without causing data races.

- **Preventing Data Races:** SafeJava provides a new static type system that prevents data races in multithreaded programs. Unlike previously proposed type systems for preventing data races, SafeJava lets programmers write generic code to implement a class, then create different objects of the class that are protected differently from data races. For example, in SafeJava, programmers can write a generic Queue implementation, then create different Queue objects that can include:

  – Queue objects protected by mutual exclusion locks, containing
    - Queue items protected by mutual exclusion locks
    - Queue items encapsulated within other data structures

  – Thread-local Queue objects,

  – Queue objects with unique pointers, and

  – Immutable Queue objects, containing
    - Queue items protected by mutual exclusion locks
    - Thread-local Queue items
    - Queue items encapsulated within other data structures

  In previous type systems, one needed a different Queue implementation to support each of the above cases.

- **Preventing Deadlocks:** SafeJava provides a new static type system that prevents deadlocks in multithreaded programs. It makes the following contributions:

  Static Lock Levels: SafeJava allows programmers to partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order.

  Lock Level Polymorphism: SafeJava allows programmers write code that is polymorphic in lock levels. SafeJava also allows programmers to specify a partial order among formal lock level parameters using where clauses [50, 112]. This enables programmers to write code in which the exact levels of some locks are not known statically, but only some ordering constraints among the unknown lock levels are known statically.

  Support for Condition Variables: In addition to mutual exclusion locks, SafeJava prevents deadlocks in the presence of condition variables. SafeJava statically enforces the constraint that a thread can invoke e.wait only if the thread holds no locks other than the lock on e. Since a thread releases the lock on e on executing e.wait, the above constraint implies that any thread that is waiting on a condition variable holds no locks. This in turn implies that there cannot be a deadlock that involves a condition variable. SafeJava thus prevents the nested monitor problem [105].

  Partial-Orders Based on Mutable Trees: SafeJava allows programmers to use recursive tree-based data structures to further order the locks within a given lock level. SafeJava allows mutations that change the partial order at runtime. The type checker uses an intraprocedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks.

  Partial-Orders Based on Monotonic DAGs: SafeJava also allows programmers to use recursive DAG-based data structures to order the locks within a given lock level. DAG edges cannot be modified once initialized. Only newly created nodes may be added to a DAG by initializing the newly created nodes to contain DAG edges to existing DAG nodes.

Runtime Ordering of Locks: SafeJava supports imposing an arbitrary linear order at runtime on locks within a given lock level. SafeJava also provides a primitive to acquire such locks in the linear order.

- **Enabling Safe Software Upgrades in Persistent Object Stores:** SafeJava enables software upgrades in persistent object stores to be defined modularly and implemented efficiently. It makes the following contributions:

  Defining Upgrade Modularity Conditions: This thesis defines *upgrade modularity conditions* that any upgrade system must satisfy to support local reasoning about upgrades. These conditions are more general than earlier definitions [138]: they apply to both lazy and stop-the-world upgrade systems; they also apply to both systems that use versions and systems that don't.

  Satisfying Upgrade Modularity Conditions: The thesis then describes a new approach for executing upgrades efficiently while satisfying the upgrade modularity conditions. The approach exploits object encapsulation properties in a novel way. The thesis proves that our upgrade system satisfies the upgrade modularity conditions when transforms are well-behaved. The thesis also shows that the conditions hold through the use of triggers and versions.

- **Enabling Safe Region-Based Memory Management:** SafeJava allows programs to safely manage their own memory using regions. It makes several important technical contributions over previous type systems for region-based memory management:

  Region types for object-oriented programs: SafeJava combines region types and ownership types in a unified type system framework that statically enforces object encapsulation as well as enables safe region-based memory management.

  Region types for multithreaded programs: SafeJava introduces 1) *subregions* within a shared region, so that long-lived threads can share objects without using the heap and without memory leaks and 2) *typed portal fields* to serve as a starting point for typed inter-thread communication. It also introduces user-defined *region kinds* to support subregions and portals.

  Region types for real-time programs: SafeJava allows programs to create LT (Linear Time) and VT (Variable Time) regions as in the Real-Time Specification for Java (RTSJ). It checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions, so that they do not wait for unbounded amounts of time. It also prevents an RTSJ *priority inversion* problem.

- **Supporting Safe Runtime Downcasts With Ownership Types:** SafeJava is primarily a static type system. The type checker uses the ownership type annotations to statically ensure the absence of certain classes of errors, but it is usually unnecessary to preserve the ownership information at runtime. However, languages like Java are not purely statically typed languages. Java allows downcasts that are checked at runtime. To support safe runtime downcasts in a language with ownership types, the system must preserve some ownership information at runtime.

  This thesis describes an efficient technique for supporting safe runtime downcasts in SafeJava. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. Moreover, this technique does not use any interprocedural analysis, so it preserves the separate compilation model of Java.

- **Type Inference:** Although SafeJava is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information. Instead, SafeJava uses a combination of type inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. SafeJava also supports user-defined defaults to cover specific patterns that might occur in user code. Note that our approach to inference is purely intraprocedural and we do not infer method signatures or types of instance variables. Rather, we use a default completion of partial type specifications in those cases. This approach permits separate compilation.

- **Programming Experience:** To gain preliminary experience, we implemented several Java programs in SafeJava. These include classes from the Java libraries, multithreaded Java server programs, and Real-Time Specification for Java (RTSJ) programs. These programs exhibit a variety of programming paradigms. We found that SafeJava is expressive enough to support these programs.

  In each case, once we understood how the program worked, adding the extra type annotations was fairly straightforward. On average, we had to annotate about one in thirty lines of code.

1.7 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the basic SafeJava type system for enforcing object encapsulation. Chapter 3 extends the type system to combine object encapsulation with effects clauses, unique pointers, and immutable objects. Chapter 4 builds on this type system to prevent data races and deadlocks in multithreaded programs. This chapter also introduces our type inference techniques that reduce programming overhead. We also describe how SafeJava supports safe runtime downcasts in this chapter. Chapter 5 shows how the type system enables safe software upgrades in persistent object stores. Chapter 6 extends the system to prevent memory errors in programs that manage their own memory using regions. Chapter 7 concludes.

Chapter 2

Enforcing Object Encapsulation

The ability to reason locally about program correctness is crucial when dealing with large programs. Local reasoning allows correctness to be dealt with one module at a time. The standard approach is to provide each module with a specification describing its expected behavior. The goal is to prove that each module satisfies its specification, using only the specifications but not code of other modules. This way the complexity of the proof effort (formal or informal) can be kept under control.

This local reasoning approach is sound if separate verification of individual modules suffices to ensure the correctness of the composite program [98, 56]. The key to sound local reasoning in object-oriented languages is object encapsulation. Consider, for example, a Stack object s that is implemented using a linked list. Local reasoning about the correctness of the Stack implementation is possible if objects outside s do not directly access the list nodes, i.e., the list nodes are *encapsulated* within the s.

SafeJava uses a variant of ownership types for specifying and statically enforcing object encapsulation. In SafeJava, a program can declare that s *owns* all the list nodes. The type system then statically ensures that the list nodes are encapsulated within s.

A type system that strictly enforces object encapsulation, however, is too constraining [113]: it does not allow efficient implementation of important constructs like iterators [104, 71]. Consider, for example, an iterator over the above-mentioned Stack object s. If the iterator is encapsulated within s, it cannot be used outside s. If the iterator is *not* encapsulated within s, it cannot directly access the list nodes in s, and hence cannot run efficiently.

Previous ownership type systems were either too constraining to support constructs like iterators [43, 42], or too permissive to support local reasoning [41]; e.g., they allowed objects outside the above-mentioned Stack object s to temporarily get direct access to the list nodes.

This thesis argues that the right way to solve the problem is to provide special access privileges to objects belonging to classes in the same module; we show how to do this for inner classes [107, 89]. SafeJava allows inner class objects to have privileged access to the representations of the corresponding outer class objects. This principled violation of encapsulation allows programmers to express constructs like iterators using inner classes, yet supports local reasoning about the correctness of the classes. SafeJava supports local reasoning because a class and its inner classes can be reasoned about together as a module.

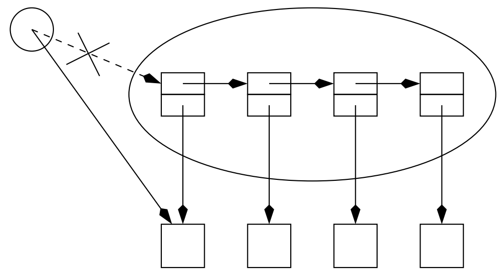

Figure 2-1: Stack Object With Encapsulated Linked List

The rest of this chapter is organized as follows. Section 2.1 discusses object encapsulation. Section 2.2 introduces the basic SafeJava type system that uses ownership types to enforce object encapsulation. Section 2.3 provides a formal description of the type system. Section 2.4 describes extensions to the type system to support inner classes. Section 2.5 discusses some practical issues about SafeJava. Section 2.6 presents related work, and Section 2.7 concludes.

2.1 Object Encapsulation

Reasoning about a class in an object-oriented program involves reasoning about the behavior of objects belonging to the class. Typically objects point to other *subobjects*, which are used to represent the containing object. Local reasoning about class correctness is possible if the subobjects are *encapsulated*:

E1. An object x *encapsulates* an object y if it maintains an encapsulation boundary such that y is inside the boundary and furthermore if z is outside the boundary, then z cannot access y. (An object z *accesses* an object y if z has a pointer to y, or methods of z obtain a pointer to y.)

Encapsulation of subobjects supports local reasoning because it ensures that outside objects cannot interact with the subobjects without calling methods of the containing object. And therefore the containing object is in control of its subobjects.

However, encapsulation of all subobjects is often more than is needed for local reasoning. Encapsulation is only required for subobjects that the containing object *depends on* [98]:

E2. An object a *depends on* subobject b if a reads/writes fields of b or calls methods of b and furthermore these reads/writes or calls expose mutable behavior of b in a way that affects the invariants of a.

Thus, a stack implemented using a linked list depends on the list but not on the items contained in the list. If code outside could manipulate the list, it could invalidate the correctness of the stack implementation. But code outside can safely use the items contained in the stack because the stack doesn't call their methods; it only depends on the identities of the items and the identities never change. Similarly, a set of immutable elements does not depend on the elements even if it invokes `a.equals(b)` to ensure that no two elements `a` and `b` in the set are equal, because the elements are immutable.

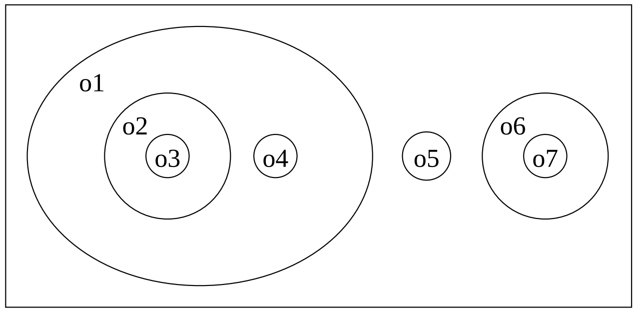

Figure 2-2: An Encapsulation Relation

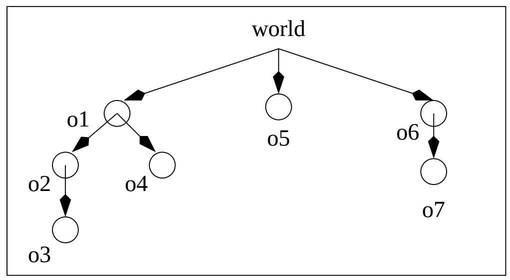

Figure 2-3: An Ownership Relation

We therefore propose the following methodology:[1]

E3. Every object must encapsulate all objects it depends on.

Figure 2-1 shows an example, where a stack object is implemented using a linked list. Since the stack object depends on the list nodes, the list nodes are encapsulated within the stack object and outside objects cannot directly access the list nodes. But stack object does not depend on the items stored in the stack, so the items stored in the stack are not encapsulated in the stack object.

In general, the encapsulation relation forms a hierarchy. In Figure 2-2, objects o2, o3, and o4 are encapsulated within o1, object o3 is encapsulated within o2, and object o7 is encapsulated within o6.

2.2 Ownership Types for Enforcing Object Encapsulation

Ownership types [23, 24, 26, 41, 43] provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects it depends on, thus preventing them from being accessible outside. This section presents the basic SafeJava type system.

2.2.1 Object Ownership

The key to the type system is the concept of object ownership. Every object has an owner. The owner can either be another object or a special owner called world.

[1] We relax this methodology slightly in Section 2.4 to support constructs like iterators.

SJ1. Every object has an owner.

SJ2. The owner can either be another object or world.

SJ3. The ownership relation forms a tree rooted at world.

SJ4. The owner of an object does not change over time.
(Except if there is a unique pointer to that object. See Section 3.3.)

SJ5. If object z owns y but $z \not\succeq x$, then x cannot access y.
(Except if x is an inner class object of z. See Section 2.4.)

Figure 2-4: Ownership Properties

Figure 2-3 presents an example ownership relation. We draw an arrow from x to y if x owns y. In the figure, the special owner world owns objects o1, o5, and o6; o1 owns o2 and o4; o2 owns o3; and o6 owns o7. The ownership relation in Figure 2-3 corresponds to the encapsulation relation in Figure 2-2.

Ownership allows a program to statically declare encapsulation boundaries that capture dependencies:

E4. An object should own all the objects it depends on.

The system then enforces encapsulation: if y is inside the encapsulation boundary of z and x is outside, then x cannot access y. In Figure 2-3, o7 is inside the encapsulation boundary of o6 and o1 is outside, so o1 cannot access o7. o1 can only access objects o2, o4, o5, and o6.

In general, an object can only access: 1) itself and objects it owns, 2) its ancestors in the ownership tree and objects they own, and 3) globally accessible objects, namely objects owned by world.

Note the analogy with nested procedures: proc P_1 {var x_2; proc P_2 {var x_3; proc P_3 {...}}}. Say x_{n+1} and P_{n+1} are children of P_n. Then P_n can only access: 1) P_n and its children, 2) the ancestors of P_n and their children, and 3) global variables and procedures.

We use the notation $o_1 \succeq o_2$ to denote that o_1 directly or transitively owns o_2 or if o_1 is the same as o_2. The relation $\succeq$ is thus the reflexive transitive closure of the *owns* relation.

SafeJava statically guarantees the properties in Figure 2-4. SJ3 states that our ownership relation has no cycles. SJ4 states that the owner of an object does not change over time. SJ5 states the encapsulation property of our system, that if y is inside the encapsulation boundary of z and x is outside, then x cannot access y.

(We later present extensions to the basic type system that relax Properties SJ4 and SJ5. Sections 2.4 relaxes Property SJ5 to allow inner class objects to have privileged access to the owned objects of the corresponding outer class objects. Section 3.3 shows how unique pointers can be used to support ownership transfer.)

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\mathit{defn} & ::= & \mathsf{class}\ cn\ \mathsf{extends}\ c\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
c & ::= & cn \mid \mathsf{Object} \\
\mathit{meth} & ::= & t\ mn(\mathit{arg}^*)\ \{e\} \\
\mathit{field} & ::= & t\ fd \\
\mathit{arg} & ::= & t\ x \\
t & ::= & c \mid \mathsf{int} \\
\mathit{formal} & ::= & f \\
e & ::= & x \mid x = e \mid \mathsf{let}\ (\mathit{arg}{=}e)\ \mathsf{in}\ \{e\} \mid x.fd \mid x.fd = y \mid e_{\mathrm{new}} \mid e_{\mathrm{method}} \\
e_{\mathrm{new}} & ::= & \mathsf{new}\ c \\
e_{\mathrm{method}} & ::= & x.mn(y^*) \\
\\
cn & \in & \text{class names} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
x, y & \in & \text{variable names}
\end{array}
$$

Figure 2-5: Grammar for a Core Subset of Java

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \mathsf{class}\ cn\langle \mathit{formal}+\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ \mathit{constr}^*\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
c & ::= & cn\langle \mathit{owner}+\rangle \mid \mathsf{Object}\langle \mathit{owner}\rangle \\
\mathit{owner} & ::= & \mathit{formal} \mid \mathsf{world} \mid \mathsf{this} \\
\mathit{constr} & ::= & (\mathit{owner} \succeq \mathit{owner}) \mid (\mathit{owner} \not\succeq \mathit{owner}) \\
\mathit{meth} & ::= & t\ mn\langle \mathit{formal}^*\rangle(\mathit{arg}^*)\ \mathsf{where}\ \mathit{constr}^*\ \{e\} \\
e_{\mathrm{method}} & ::= & x.mn\langle \mathit{owner}^*\rangle(y^*) \\
\\
f & \in & \text{owner names}
\end{array}
$$

Figure 2-6: Grammar Extensions to Support Ownership Types

2.2.2 Owner Polymorphism

To simplify the presentation of key ideas, we describe our type system in the context of a core subset of Java [77]. Our approach, however, extends to the whole of Java and other similar languages. Figure 2-5 presents the grammar for our Java subset. This Java subset is similar to Classic Java [70], and has much of the same semantics as Classic Java. A program is a sequence of class definitions followed by an initial expression. A predefined class Object is the root of the class hierarchy. The SafeJava language constructs are similar to the corresponding constructs in Java.

Figure 2-6 presents grammar extensions to support ownership types. Figure 2-7 shows an example TStack program. A TStack is a stack of T objects. It is implemented using a linked list. For simplicity, all examples in this thesis use a language that is syntactically close to Java. (The example shows type annotations written explicitly. However, many of them can be automatically inferred. We discuss type inference in Section 2.5.)

Every class definition is parameterized with one or more owners. The first owner parameter is special: it identifies the owner of the corresponding object. The other owner parameters propagate the ownership information. Parameterization allows programmers to implement a generic class whose objects have different owners. This parameterization is similar to parametric polymorphism [3, 29, 32, 90, 112, 132] except that our parameters are owners, not types. Our type system would fit naturally in a language with parameterized types.[2]

[2]If we had parameterized types in the language, the Stack declaration would look like the following: `class Stack<stackOwner>[T<TOwner>] {...}`

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  TNode<nodeOwner, TOwner> next;  T<TOwner> value;

  TNode(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;
    this.next  = n;
  }
  T<TOwner> value()               { return value; }
  TNode<nodeOwner, TOwner> next() { return next;  }
}

class T<TOwner> { }

class TStackClient<clientOwner> {
  void test() {
    TStack<this,  this>  s1;
    TStack<this,  world> s2;
    TStack<world, world> s3;
 /* TStack<world, this>  s4; illegal! */
}}
```

Figure 2-7: Stack of T Objects

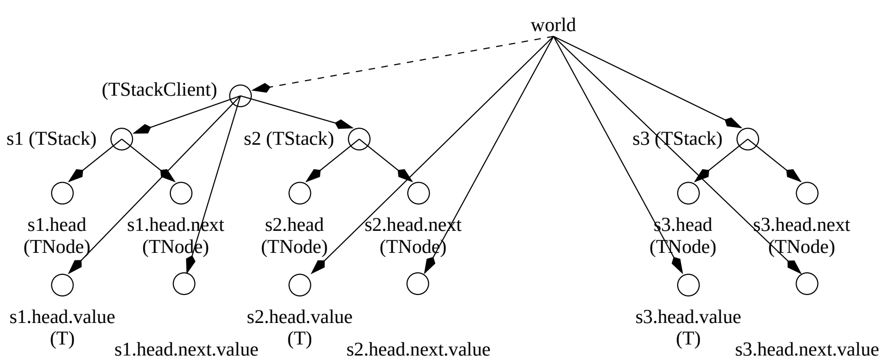

Figure 2-8: Ownership Relation for TStacks s1, s2, s3

```
class C<cOwner, sOwner, tOwner> where (tOwner >= sOwner) {
    ...
    TStack<sOwner, tOwner> s;
}
```

Figure 2-9: Using Where Clauses to Constrain Owners

An owner can be instantiated with this, with world, or with another owner parameter. Objects owned by this are encapsulated objects that cannot be accessed from outside. Objects owned by world can be accessed from anywhere.

In Figure 2-7, the TStack class is parameterized by stackOwner and TOwner. stackOwner owns the TStack object. (Recall that the first owner parameter always owns the corresponding object.) TOwner owns the T objects contained in the TStack. The code specifies that the TStack object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the TStack object.

The type of TStack s1 is instantiated using this for both the owner parameters. This means that TStack s1 is owned by the TStackClient object that created it and so are the T objects in s1. TStack s2 is owned by the TStackClient object, but the T objects in s2 are owned by world. TStack s3 is owned by world and so are the T objects in s3. The ownership relation for s1, s2, and s3 is depicted in Figure 2-8 (assuming the stacks contain two elements each). (The dotted line indicates that every object is transitively owned by world.)

2.2.3 Constraints on Owners

For every type $cn\langle o_1, ..., o_n\rangle$ with multiple owners, SafeJava statically enforces the constraint that $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. (Recall from Figure 2-4 that the ownership relation forms a tree rooted at world, and that the notation $(o_a \succeq o_b)$ means that either o_b is the same as o_a or o_b is a descendant of o_a in the ownership tree.) Thus, the type of TStack s4 in Figure 2-7 is illegal because (this $\not\succeq$ world).

The above constraint is the same as in [41]. However, we extend it to parameterized methods as well. For a method $m\langle o_{n+1}, ..., o_k\rangle(...)\{...\}$ of an object of type $cn\langle o_1, ..., o_n\rangle$, the restriction is that $(o_i \succeq o_1)$ for all $i \in \{1..k\}$.

(These constraints are necessary to provide encapsulation in the presence of subtyping. Otherwise, subtyping interacts with ownership in a subtle way to violate encapsulation. Figure 2-14 in Section 2.6 illustrates this point with an example. Also, Theorem 1 in Section 2.2.4 uses these constraints to prove the encapsulation guarantee provided by SafeJava.)

To allow ownership constraints to be checked modularly, it is sometimes necessary for programmers to specify additional constraints on class and method parameters. For example, in Figure 2-9, the type of s is legal only if (tOwner $\succeq$ sOwner). SafeJava allows programmers to specify such additional constraints using where clauses [50, 112], and it statically enforces the constraints. For example, in Figure 2-9, class C specifies that (tOwner $\succeq$ sOwner). An instantiation of C that does not satisfy the constraint is illegal.

Subtyping

The rule for declaring a subtype is that the first owner parameter of the supertype must be the same as that of the subtype; in addition, of course, the supertype must satisfy the constraints on owners. The first owners have to match because they are special, in that they own the corresponding objects. Thus, $\mathsf{TStack}\langle\mathsf{stackOwner},\ \mathsf{TOwner}\rangle$ is a subtype of $\mathsf{Object}\langle\mathsf{stackOwner}\rangle$. But $\mathsf{T}\langle\mathsf{TOwner}\rangle$ is not a subtype of $\mathsf{Object}\langle\mathsf{world}\rangle$ because the first owners do not match.

2.2.4 Encapsulation Theorem

The SafeJava type system we describe so far (without inner classes) enforces the following encapsulation property:

THEOREM 1. *In SafeJava without inner classes, an object x can access an object owned by o only if ($o \succeq x$)*

PROOF. Consider the code: *class* $C\langle f, ...\rangle\{...\ T\langle o, ...\rangle\ y\ ...\}$. Variable y of type $T\langle o, ...\rangle$ is declared within the static scope of class C. Owner o can therefore be either 1) this, or 2) world, or 3) a formal class parameter, or 4) a formal method parameter. We will show that in each case, the constraint ($o \succeq \mathsf{this}$) holds. In the first two cases, the constraint holds trivially. In the last two cases, ($o \succeq f$) and ($f \succeq \mathsf{this}$), so the constraint holds. Therefore an object x of a class C can access an object y owned by o only if ($o \succeq x$). $\square$

The above theorem is equivalent to Property SJ5 in Figure 2-4. It is easy to see that the type system described so far also preserves Properties SJ1 to SJ4.

2.3 Formal Description

The previous section presented the grammar for our basic type system in Figures 2-5 and 2-6. This section describes some of the important rules for type checking. The full set of rules are in shown in Appendix 2.A at the end of this chapter.

The core of our type system is a set of rules for reasoning about the typing judgment: $P; E \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. t is the type of e.

We define a typing environment as: $E ::= \emptyset \mid E, t\ x \mid E, \text{owner}\ f \mid E, \mathit{constr}$
The typing environment contains the declared types of variables, the declared owner parameters, and the declared constraints among owners.

A useful auxiliary judgment is $P; E \vdash \mathit{constr}$, where *constr* is an ownership constraint of the form either $o_1 \succeq o_2$ or $o_1 \not\succeq o_2$. The judgment states that the constraint holds in the typing environment E. The rules for this judgment are as follows:

[CONSTR ENV]
$$\frac{E = E_1,\ \mathit{constr},\ E_2}{P; E \vdash \mathit{constr}}$$

[OWNER $\succeq$]
$$\frac{P; E \vdash e : cn\langle o_{1..n}\rangle}{P; E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]
$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (\mathsf{world} \succeq o)}$$

[REFL $\succeq$]
$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (o \succeq o)}$$

[TRANS $\succeq$]
$$\frac{\begin{array}{c}P; E \vdash (o_3 \succeq o_2)\\ P; E \vdash (o_2 \succeq o_1)\end{array}}{P; E \vdash (o_3 \succeq o_1)}$$

The first rule states that if a constraint is part of a class or method declaration, then the type checker assumes that constraint within the scope of that class or method. The second rule states that the first owner parameter of a type owns the corresponding object. (Recall this fact from Section 2.2.2). The third rule states that world transitively owns all objects. The fourth and fifth rules state that the ownership relation is reflexive and transitive.

The rule for creating a new object of type $cn\langle o_{1..n}\rangle$ must ensure that the type $cn\langle o_{1..n}\rangle$ is valid. The rule for that checks that $cn\langle f_{1..n}\rangle$ is a valid class; that owners $o_1..o_n$ are valid owners (a valid owner is either this, or world, or a formal owner parameter); that $o_i \succeq o_1$ for all $i \in \{1..n\}$ (recall this constraint from Section 2.2.3); and that all the other constraints in the declaration of class cn are satisfied. Note that since each constraint *constr* is declared inside the class, it might contain occurrences of the formal owner parameters $f_1..f_n$. When *constr* is used outside the class, one must rename the formal parameters with their corresponding actual owner parameters $o_1..o_n$.

[EXP NEW]

$$\frac{P;\, E \vdash cn\langle o_{1..n}\rangle}{P;\, E \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle \ldots\ \mathsf{where}\ constr* \ \ldots \\ P;\, E \vdash_{\mathsf{owner}} o_i \quad P;\, E \vdash o_i \succeq o_1 \quad P;\, E \vdash constr\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E \vdash cn\langle o_{1..n}\rangle}$$

The rules for subtyping are similar to those in parametric polymorphism [3, 29, 32, 90, 112, 132], except that the first owner parameter of the supertype must be the same as that of the subtype. The first owners have to match because they are special, in that they own the corresponding objects. The subtyping relation is reflexive and transitive.

[SUBTYPE C]

$$\frac{\begin{array}{c} P;\, E \vdash cn\langle o_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ cn'\langle f_1\ o*\rangle \ \ldots \end{array}}{P;\, E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{\begin{array}{c} P;\, E \vdash t_1 <: t_2 \\ P;\, E \vdash t_2 <: t_3 \end{array}}{P;\, E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t <: t}$$

The rule for a let expression simply adds the new variable and its type to the type environment. The rules for reading and writing a variable look up the type of the variable from the type environment.

[EXP LET]

$$\frac{\begin{array}{c} arg = t\ x \quad P;\, E \vdash e : t \\ P;\, E,\, arg \vdash e' : t' \end{array}}{P;\, E \vdash \mathsf{let}\ (arg = e)\ \mathsf{in}\ \{e'\} : t'}$$

[EXP VAR]

$$\frac{E = E_1,\, t\ x,\, E_2}{P;\, E \vdash x : t}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P;\, E \vdash x : t \\ P;\, E \vdash e : t \end{array}}{P;\, E \vdash x = e : t}$$

The rule for accessing field $x.fd$ checks that x is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t. Note that the rule renames t when it used outside class cn.

The rule for assigning to a field is similar.

[EXP REF]

$$\frac{P;\, E \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle}{P;\, E \vdash x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{\begin{array}{c} P;\, E \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\, E \vdash y : t\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E \vdash x.fd = y : t\ [o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method on an object checks that the type of that object declares or inherits the method, and that the method arguments are of the right type. Since methods in our system may be parameterized, the rule checks that the method parameters are instantiated with valid owners, and that the constraints on the owners satisfied. The rule appropriately renames all the types when they are used outside their declared context.

[EXP INVOKE]

$$\frac{\begin{array}{c} P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \textsf{where}\ constr*\ \{e\}) \in cn\langle f_{1..n}\rangle \\ P; E \vdash x : cn\langle o_{1..n}\rangle \quad P; E \vdash o_i \succeq o_1 \quad P; E \vdash constr\ [o_1/f_1]..[o_m/f_m] \\ P; E \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \end{array}}{P; E \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

2.4 Inner Classes

The previous sections presented the basic SafeJava type system that enforces object encapsulation and thus enables local reasoning about program correctness. But strict object encapsulation is too constraining [113]: it prevents efficient implementation of important constructs like iterators. For example, to run efficiently, an iterator over a Stack object s implemented using a linked list needs access to the list nodes in s. To provide this access, we have to allow objects like iterators to violate encapsulation.

Local reasoning is still possible provided all violations of encapsulation are limited to code contained in the same module. For example, if both the Stack and its iterator are implemented in the same module, one can still reason about their correctness locally, by examining the code of that module.

This section extends our basic type system with inner classes to support constructs like iterators, while supporting local reasoning. Our inner classes are similar to the member inner classes in Java. Inner class definitions are nested inside other classes. Figure 2-10 shows grammar extensions to support inner classes. Figure 2-11 shows an example. In the figure, the inner class TStackEnum implements an iterator for the TStack; the elements method of TStack provides a way of creating an iterator over the TStack. The TStack code is otherwise similar to that in Figure 2-7.

An inner class is parameterized with owners like a regular class. In Figure 2-11, the TStack-Enum class is parameterized with enumOwner. Its complete type is TStack⟨stackOwner, TOwner⟩.TStackEnum⟨enumOwner⟩. It is declared to be a subtype of TEnumeration⟨enum-Owner, TOwner⟩. Like a regular class, the first owner parameter of an inner class identifies the owner of the corresponding object. In Figure 2-11, enumOwner owns the TStackEnum object.

Recall from before that for every type $cn\langle o_1, ..., o_n\rangle$ with multiple owners, SafeJava statically enforces the constraint that $(o_i \succeq o_1)$ for all valid i. For an inner class cn_1 of type $cn_k\langle o_{k1..kn_k}\rangle..cn_2\langle o_{21..2n_2}\rangle.cn_1\langle o_{11..1n_1}\rangle$, the constraint is that $(o_{ij} \succeq o_{11})$ for all valid i, j. (This constraint is necessary to support local reasoning.) Thus, in Figure 2-11, it must be the case that (TOwner $\succeq$ enumOwner) and (stackOwner $\succeq$ enumOwner).

The elements method in the figure is parameterized by enumOwner. It can only be instantiated with an owner that satisfies the above mentioned constraints, that (TOwner $\succeq$

$$
\begin{array}{rcl}
\textit{defn} & ::= & \textsf{class}\ cn\langle \textit{formal+}\rangle\ \textsf{extends}\ c\ \textsf{where}\ \textit{constr*}\ \{\textit{innerclass*}\ \textit{field*}\ \textit{meth*}\} \\
\textit{innerclass} & ::= & \textit{defn} \\
c & ::= & cn\langle \textit{owner+}\rangle \mid \textsf{Object}\langle \textit{owner}\rangle \mid c.cn\langle \textit{owner+}\rangle \\
\textit{owner} & ::= & \textit{formal} \mid \textsf{world} \mid \textsf{this} \mid cn.\textsf{this} \\
e_{\text{new}} & ::= & \textsf{new}\ c \mid x.\textsf{new}\ c
\end{array}
$$

Figure 2-10: Grammar Extensions to Support Inner Classes

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<enumOwner> elements<enumOwner>()
    where (TOwner >= enumOwner) (stackOwner >= enumOwner) {
    return new TStackEnum<enumOwner>();
  }

  class TStackEnum<enumOwner> implements TEnumeration<enumOwner, TOwner> {
    TNode<TStack.this, TOwner> current;

    TStackEnum() { current = TStack.this.head; }

    T<TOwner> getNext() {
      if (current == null) return null;
      T<TOwner> t = current.value();  current = current.next(); return t;
    }
    boolean hasMoreElements() { return (current != null); }
}}

class TStackClient<clientOwner> {
  void test() {
    TStack<this, world>        s = new TStack<this, world>();
    TEnumeration<this,  world> e = s.elements<this>();
}}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext();
  boolean   hasMoreElements();
}
```

Figure 2-11: TStack Iterator

enumOwner) and (stackOwner $\succeq$ enumOwner). This requirement is captured in the where clause. Note that the constraints on method parameters described in Section 2.2.3 also imply that for iterators created with the elements method, it must be that (enumOwner $\succeq$ stackOwner). Thus, for iterators used outside the TStack class, enumOwner and stackOwner must be the same.

Recall also that in a regular class, an owner can be instantiated with this, with world, or with another owner parameter. Within an inner class, an owner can also be instantiated with cn.this, where cn is an outer class. This feature allows an inner object to access the objects encapsulated within its outer objects. In Figure 2-11, the owner of the current field inTStackEnum is instantiated with TStack.this. The current field accesses list nodes encapsulated within its outer TStack object.

Encapsulation Theorem

The SafeJava type system (with inner classes) provides the following encapsulation property:

THEOREM 2. *An object x can access an object owned by o only if:*

1. *$(o \succeq x)$, or*
2. *x is an inner class object of o.*

PROOF. Consider the code: *class* $C\langle f, ...\rangle\{...\ T\langle o, ...\rangle\ y\ ...\}$. Variable y of type $T\langle o, ...\rangle$ is declared within the static scope of class C. Owner o can therefore be either 1) this, or 2) world, or 3) a formal class parameter, or 4) a formal method parameter, or 5) C'.this, where C' is an outer class. We will show that in the first four cases, the constraint ($o \succeq$ this) holds. In the first two cases, the constraint holds trivially. In the last two cases, $(o \succeq f)$ and ($f \succeq$ this), so the constraint holds. In the fifth case, (C'.this $= o$). Therefore an object x of a class C can access an object y owned by o only if either 1) $(o \succeq x)$, as in the first four cases, or 2) x is an inner object of o, as in the fifth case. □

Discussion

In a previous version of our system [24], the typing rules for inner classes were more flexible. An inner class was parameterized with owners just like a regular class. However, the outer class parameters were not automatically visible inside an inner class. If an inner class used an outer class parameter, it had to explicitly include the outer class parameter in its declaration. For example, consider the TStackEnum class in Figure 2-12 which is implemented in the previous version of our type system. The TStackEnum declaration includes the owner parameter TOwner from its outer class. TOwner is therefore visible inside TStackEnum. But the TStackEnum declaration does not include stackOwner. Therefore, stackOwner is not visible inside TStackEnum.

Recall that for an inner class cn_1 of type $cn_k\langle o_{k1..kn_k}\rangle..cn_2\langle o_{21..2n_2}\rangle.cn_1\langle o_{11..1n_1}\rangle$, SafeJava enforces the constraint that $(o_{ij} \succeq o_{11})$ for all valid i, j. The previous version of the system only enforced the constraint that $(o_{1j} \succeq o_{11})$ for all valid j.

This additional flexibility in typing rules allowed programmers to create wrappers [71] that exposed a limited interface to an underlying object, and use the wrappers in contexts where the underlying objects were inaccessible. In the figure, the TStackClient creates an unencapsulated iterator e2 over an encapsulated TStack s; the program can then pass e2 to objects outside the TStackClient. In general, the typing rules in the previous version of the type system allowed programs to use an inner class to create a wrapper around an encapsulated subobject and pass the wrapper object outside the encapsulation boundary.

Even though the previous version of the type system was more expressive, we encountered problems when we were extending it to ensure safe memory management and safe multithreading. For example, for safe region-based memory management, we had to ensure that the region containing an inner class object does not outlive the region containing its outer class object, because otherwise that could create dangling references. Similarly, for safe multithreading, we had to ensure that whenever a thread accesses a shared outer class object through its inner class object, that the thread first acquires the lock that protects the outer class object, because otherwise that could cause data races. To check these conditions statically, we had to ensure that an inner class object is not accessible outside the encapsulation boundary where the outer class object is inaccessible.

We therefore took the current approach in SafeJava.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<enumOwner, TOwner> elements<enumOwner>()
    where (TOwner >= enumOwner) {
    return new TStackEnum<enumOwner, TOwner>();
  }

  class TStackEnum<enumOwner, TOwner> implements TEnumeration<enumOwner, TOwner> {
    TNode<TStack.this, TOwner> current;

    TStackEnum() { current = TStack.this.head; }

    T<TOwner> getNext() {
      if (current == null) return null;
      T<TOwner> t = current.value();  current = current.next(); return t;
    }
    boolean hasMoreElements() { return (current != null); }
}}

class TStackClient<clientOwner> {
  void test() {
    TStack<this, world>       s  = new TStack<this, world>();
    TEnumeration<this,  world> e1 = s.elements<this> ();
    TEnumeration<world, world> e2 = s.elements<world>();
}}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext();
  boolean   hasMoreElements();
}
```

Figure 2-12: TStack Iterator in a Previous Version of our System [24]

2.5 Practical Issues

Although SafeJava is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information. Instead, SafeJava uses a combination of inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. We describe type inference and default types later in the thesis, in Section 4.5. We emphasize that this approach to inference is purely intraprocedural and does not infer method signatures or types of instance variables. Rather, it uses a default completion of partial type specifications in those cases to minimize the required annotations. This approach permits separate compilation. On average, we had to change about one in thirty lines of code to express Java programs in SafeJava. We describe programming experience in SafeJava in Sections 4.8 and 6.7.

The system we described is a purely static type system. The ownership relations are used for compile-time type checking and are not preserved at runtime. Consequently, SafeJava programs have no runtime overhead compared to regular Java programs. In fact, one way to compile and run a SafeJava program is to convert it to a Java program after type checking, by removing the owner parameters, the constraints on owners, and the effects clauses.

A language like Java, however, is not purely statically typed. Java allows downcasts that are checked at runtime. Suppose an object with declared type $\mathsf{Object}\langle\mathsf{o}\rangle$ is downcast to $\mathsf{Vector}\langle\mathsf{o,e}\rangle$. Since the result of this operation depends on information that is only available at runtime, our type checker cannot verify at compile-time that e is the right owner parameter even if we assume that the object is indeed a Vector. To safely support downcasts, a system has to keep some ownership information at runtime. This is similar to keeping runtime information with parameterized types [112, 132]. In Section 4.4, we describe how to do this efficiently for ownership by keeping runtime information only for objects that can be potentially involved in downcasts into types with multiple parameters.

Our type checking rules ensure that for a program to be well-typed, the program respects the properties described in Figures 2-4 and 3-5. A complete syntactic proof [137] of type soundness can be constructed by defining an operational semantics (by extending the operational semantics of Classic Java [70]) and proving that well-typed programs do not reach an error state and the generalized subject reduction theorem holds for well-typed programs. The subject reduction theorem states that the semantic interpretation of a term's type is invariant under reduction. The proof is straightforward but tedious, so it is omitted here.

2.6 Related Work

SafeJava uses a variant of ownership types to statically enforce object encapsulation and enables local reasoning about program correctness. As we stated in Theorem 2 in Section 2.4, SafeJava provides the following encapsulation guarantee:

O1. **Owners as encapsulating objects:** An object x can access an object owned by o only if ($o \succeq x$) or x is an inner class object of o.

This section presents an overview of ownership type systems and the encapsulation guarantees they provide, and other related type systems.

Early Work:
Euclid [94] is one of the first languages that considered the problem of aliasing. [87] stressed the need for better treatment of aliasing in object-oriented programs. Early work on Islands [86] and Balloons [7] focused on *fully encapsulated* objects where all subobjects an object can access are not accessible outside the object. Universes [111] also enforces full encapsulation, except for read-only references. However, full encapsulation significantly limits expressiveness, and is often more than is needed. The work on ESC/Java pointed out that encapsulation is required only for subobjects that the containing object *depends* on [98, 56], but ESC/Java was unable to always enforce encapsulation.

Ownership Types:
Ownership type systems use naming to enforce encapsulation. The type of an object includes the name of its owner. To access an object, a program must name the type of that object, and hence must name the owner of that object. Ownership types were proposed in [43] and formalized in [42]. These systems enforce object encapsulation, but do so by significantly limiting expressiveness. In these systems, a subtype must have the same owners as a supertype. So TStack⟨thisOwner,TOwner⟩ cannot be a subtype of Object⟨thisOwner⟩. Moreover, they do not support constructs like iterators.

PRFJ, SCJ, and JOE:
PRFJ [26], SCJ [23], and JOE [41] extend ownership types to support a natural form of subtyping that is similar to subtyping in parametric type systems [3, 29, 32, 90, 112, 132].

A subtype can have different owners than a supertype. However, the first owners must match because the first owners own the corresponding object. To support subtyping, JOE enforces the constraint that in every type $T\langle o_1, ..., o_n\rangle$ with multiple owners, $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. Without this constraint and with subtyping, JOE would not have provided any meaningful encapsulation guarantees. Figure 2-14 illustrates point this with an example. PRFJ and SCJ allow an object to contain pointers to subobjects owned by a different object, but they have effects clauses that prevent a program from following such pointers. The above systems effectively enforce encapsulation for object fields. However, to support constructs like iterators, they allow method local variables to violate encapsulation. Therefore they do not support local reasoning.

Figure 2-13 presents example code in these systems that violates object encapsulation. (We adopted the example from the JOE paper [41]. But we present this and other examples in our syntax, that is slightly different from the syntax in the original papers.) The example shows an iterator for the TStack in Figure 2-7. In the example, the TStack object owns the iterator object. But a TStackClient object that is outside the encapsulation boundary of the TStack object accesses the iterator object, thus violating object encapsulation (Property O1). However, note that type of the iterator contains the TStack object. So the TStackClient object can access the iterator only when the TStack object is in scope. This ensures that the violation of object encapsulation is temporally bounded.

PRFJ, SCJ, and JOE enforce the weak encapsulation property O2. JOE also enforces the weak encapsulation property O3. These properties imply that an application must access the owner of an object before it can access the object.

- O2. **Owners as capabilities:** The owner of object x must be in scope when an application accesses x.
- O3. **Owners as dominators:** All paths in the heap from the root object to object x must pass through x's owner.

AliasJava:
AliasJava [6] uses ownership types to aid program understanding. Like other ownership type systems, AliasJava allows programmers to use ownership information to reason about aliasing. AliasJava is also more flexible than other ownership type systems. For example, in AliasJava, an iterator object that accesses encapsulated subobjects of a collection can outlive the collection object. However, unlike other ownership type systems, AliasJava does not enforce properties like O1, O2, or O3 which either disallow violations of object encapsulation entirely or temporally limit such violations. This is because AliasJava has subtyping, but it neither has the constraint that other owners $\succeq$ the first owner as in JOE [41], nor does it have effects clauses as in PRFJ [26] and SCJ [23].

Figure 2-14 presents AliasJava code that violates O1, O2, and O3. (Again, the syntax in the original paper is slightly different.) In the example, SomeClass passes its encapsulated object f to a publicly accessible object s, leading to a violation of object encapsulation (Property O1). The interaction between subtyping and ownership enables the creation of a path to f through s that does not go through f's owner. Other parts of the program can then access f using this path even if they have no relationship with f's owner. The decoupling of f from its owner is further illustrated by the fact that the program can access f even after f's owner becomes garbage.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<this, TOwner> elements() {
    return new TStackEnum<this, TOwner>(head);
  }
}
class TStackEnum<enumOwner, TOwner> {
  TNode<enumOwner, TOwner> curr;
  TStackEnum(TNode<enumOwner, TOwner> head)  {curr = head;}
  T<TOwner> getNext() {...} boolean hasMoreElements() {...}
}
class TStackClient<clientOwner> {
  void test() {
    TStack<this,  this> s = new TStack<this, this>;
    TStackEnum<s, this> e = s.elements();        /* Violates OE1 */
  }                                              /* owner of e is a local variable! */
}
```

Figure 2-13: Violation of Object Encapsulation in [23], [26], and [41]

Confined Types:
Confined types [17, 81] provide a light weight mechanism to statically ensure that objects of package protected classes do not escape their package boundaries. Confined types thus offer package level protection, unlike ownership types where the confinement is at an object level. Moreover, confined types are not polymorphic. Therefore, unlike ownership types, a program cannot create some confined and some unconfined objects of the same class.

Types for Safe Region-Based Memory Management:
Region types [39, 48, 80, 84, 128] statically ensure memory safety in programs that manage their own memory using regions. Ownership types and region types are related. Consider, for example, a Stack object s that is implemented using a Vector object v. In an object-oriented language that only has region types (e.g., [39]), the types of s and v would declare that they are allocated in some region r. In an object-oriented language that only has ownership types, the type of v would declare that it is owned by s. Chapter 6 shows how SafeJava combines regions types and ownership types in a unified type system framework. SafeJava provides a simple unified mechanism to declare *both* properties. The type of s can declare that it is allocated in r and the type of v can declare that it is owned by s. SafeJava then statically ensures that both objects are allocated in r, that there are no pointers to v and s after r is deleted, and that v is encapsulated within s.

Types for Safe Concurrent Programming:
Ownership types are related to type systems for statically preventing data races and deadlocks in multithreaded programs [68, 26, 23] because the lock that protects a shared mutable object from concurrent accesses also protects its encapsulated subobjects. Chapter 4 shows how SafeJava combines these type systems in a unified framework.

Types for Safe Software Upgrades:
Ownership types are also related to software upgrades. Chapter 5 shows how the SafeJava type system enables safe software upgrades in persistent object stores.

```
class Foo<o> { int x = 0;  void accessMe() { x++; } }

class SuperType<o> { void some_method() {} }

class SubType<o,c> extends SuperType<o> {
  Foo<c>              owner_parameter_c_owns_me;
  SubType(Foo<c> x)  {owner_parameter_c_owns_me = x;}
  void some_method() {owner_parameter_c_owns_me.accessMe();}
}
class SomeClass<o> {
  Foo<this>        f = new Foo<this>;
  SuperType<world> s = new SubType<world,this>(f);    // "this" is not >= "world" !!!
  SuperType<world> get() {return s;}
}
class Main<o> {
  void m() {
    SuperType<world> s = null;
   {SomeClass<this>  c = new SomeClass<this>; s = c.get();}
    s.some_method();                                           // Violates O1, O2, O3
}}

// SubType s is not encapsulated within SomeClass but some_method of SubType
// accesses Foo object owned by SomeClass                            Therefore Violates O1

// some_method accesses owner_parameter_c_owns_me whose owner c is now garbage
//                                                                   Therefore Violates O2

// There is path to owner_parameter_c_owns_me through s that does not go through c
//                                                                   Therefore Violates O3
```

Figure 2-14: Violation of Encapsulation in [6]

Shape Analysis:
Systems such as TVLA [120], PALE [109], and Roles [93] specify the shape of a local object graph in more detail than ownership types. TVLA can verify properties such as when the input to the program is a tree, the output is also a tree. PALE can verify all the data structures that can be expressed as graph types [91]. Roles can verify global properties such as the participation of objects in multiple data structures. Roles also support compositional interprocedural analysis. In contrast to these systems that take exponential time for verification, ownership types provide a lightweight and practical way to constrain aliasing.

2.7 Conclusions

Object encapsulation is key to sound local reasoning in object-oriented languages. This chapter presented SafeJava, the first ownership type system that can express constructs like iterators while also supporting local reasoning. The subsequent chapters of this thesis build on this type system to verify several program properties statically.

2.A Rules for Type Checking

This section formally presents the basic SafeJava type system that includes ownership types. The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & \textit{defn* e} \\
\textit{defn} & ::= & \textsf{class}\ cn\langle \textit{formal+}\rangle\ \textsf{extends}\ c\ \textsf{where}\ \textit{constr*}\ \{\textit{field* meth*}\} \\
c & ::= & cn\langle \textit{owner+}\rangle \mid \textsf{Object}\langle \textit{owner}\rangle \\
\textit{owner} & ::= & \textit{formal} \mid \textsf{world} \mid \textsf{this} \\
\textit{constr} & ::= & (\textit{owner} \succeq \textit{owner}) \mid (\textit{owner} \not\succeq \textit{owner}) \\
\textit{meth} & ::= & t\ mn\langle \textit{formal*}\rangle(\textit{arg*})\ \textsf{where}\ \textit{constr*}\ \{e\} \\
\textit{field} & ::= & t\ \textit{fd} \\
\textit{arg} & ::= & t\ x \\
t & ::= & c \mid \textsf{int} \\
\textit{formal} & ::= & f \\
 & & \\
e & ::= & \textsf{new}\ c \mid x \mid x = e \mid \textsf{let}\ (\textit{arg}{=}e)\ \textsf{in}\ \{e\} \mid x.\textit{fd} \mid x.\textit{fd} = y \mid x.mn\langle \textit{owner*}\rangle(y*) \\
 & & \\
cn & \in & \text{class names} \\
\textit{fd} & \in & \text{field names} \\
mn & \in & \text{method names} \\
x,y & \in & \text{variable names} \\
f & \in & \text{owner names}
\end{array}
$$

We first define a number of predicates used in the type system. These predicates are based on similar predicates from [70]. We refer the reader to that paper for their precise formulation.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
FieldsOnce(P)	No class contains two fields, declared or inherited, with same name
MethodsOncePerClass(P)	No class contains two methods with same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden.

The core of our type system is a set of rules for reasoning about the typing judgment:
$P;\, E \vdash e : t$

P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. t is the type of e.

We define a typing environment as $E ::= \emptyset \mid E,\, t\ x \mid E,\, \text{owner}\ f \mid E,\, \textit{constr}$

The typing environment contains the declared types of variables, the declared owner parameters, and the declared constraints among owners.

We define the type system using the following judgments. We present the typing rules for these judgments after that.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn}$	defn is a well-formed class
$P; E \vdash_{\text{owner}} o$	o is an owner
$P; E \vdash \mathit{constr}$	constraint constr is satisfied
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash wf$	typing environment E is well-formed
$P \vdash \mathit{field} \in c$	class c declares/inherits field
$P \vdash \mathit{meth} \in c$	class c declares/inherits meth
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash e : t$	expression e has type t

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c}\mathit{WFClasses(P)} \quad \mathit{ClassOnce(P)} \quad \mathit{FieldsOnce(P)} \\ \mathit{MethodsOncePerClass(P)} \quad \mathit{OverridesOK(P)} \\ P = \mathit{defn}_{1..n}\ e \quad P \vdash \mathit{defn}_i \quad P; \emptyset; \mathsf{world}; \mathsf{world} \vdash e : t\end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]

$$\frac{\begin{array}{c}E = cn\langle f_{1..n}\rangle\ \mathsf{this},\ \mathsf{owner}\ f_{1..n},\ f_i \succeq f_1,\ \mathit{constr}* \\ P; E \vdash wf \qquad P; E \vdash c' \\ P; E \vdash \mathit{field}_i \qquad P; E \vdash \mathit{meth}_i\end{array}}{\begin{array}{l}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c'\ \mathsf{where}\ \mathit{constr}* \\ \qquad\qquad\qquad\qquad \{\mathit{field}*\ \mathit{meth}*\}\end{array}}$$

$\boxed{P; E \vdash \mathit{constr}}$

[CONSTR ENV]

$$\frac{E = E_1,\ \mathit{constr},\ E_2}{P; E \vdash \mathit{constr}}$$

[OWNER $\succeq$]

$$\frac{P; E \vdash e : cn\langle o_{1..n}\rangle}{P; E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (\mathsf{world} \succeq o)}$$

[REFL $\succeq$]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (o \succeq o)}$$

[TRANS $\succeq$]

$$\frac{\begin{array}{c}P; E \vdash (o_3 \succeq o_2) \\ P; E \vdash (o_2 \succeq o_1)\end{array}}{P; E \vdash (o_3 \succeq o_1)}$$

$\boxed{P; E \vdash_{\text{owner}} o}$

[OWNER WORLD]

$$\frac{}{P; E \vdash_{\text{owner}} \mathsf{world}}$$

[OWNER FORMAL]

$$\frac{E = E_1,\ \mathsf{owner}\ f,\ E_2}{P; E \vdash_{\text{owner}} f}$$

[OWNER THIS]

$$\frac{E = E_1,\ c\ \mathsf{this},\ E_2}{P; E \vdash_{\text{owner}} \mathsf{this}}$$

$\boxed{P; E \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P; \emptyset \vdash wf}$$

[ENV X]

$$\frac{\begin{array}{c}P; E \vdash t \\ x \notin \mathrm{Dom}(E) \\ P; E \vdash wf\end{array}}{P; E, t\ x \vdash wf}$$

[ENV OWNER]

$$\frac{\begin{array}{c}f \notin \mathrm{Dom}(E) \\ P; E \vdash wf\end{array}}{P; E,\ \mathsf{owner}\ f \vdash wf}$$

[ENV CONSTR]

$$\frac{\begin{array}{c}\mathit{constr} = (o' \succeq o) \vee \mathit{constr} = (o' \not\succeq o) \\ P; E \vdash wf \quad\ P; E \vdash_{\text{owner}} o, o' \\ E' = E,\ \mathit{constr} \\ \nexists_{x,y}\ (P; E' \vdash y \succeq x) \wedge (P; E' \vdash y \not\succeq x)\end{array}}{P; E,\ \mathit{constr} \vdash wf}$$

$\boxed{P; E \vdash t}$

[TYPE INT]

$$\frac{}{P; E \vdash \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \mathsf{where}\ \mathit{constr}* \ ... \\ P; E \vdash_{\text{owner}} o_i \quad P; E \vdash o_i \succeq o_1 \quad P; E \vdash \mathit{constr}\ [o_1/f_1]..[o_n/f_n]\end{array}}{P; E \vdash cn\langle o_{1..n}\rangle}$$

$\boxed{P;\, E \vdash t_1 <: t_2}$

[SUBTYPE C]

$$\frac{P;\, E \vdash cn\langle o_{1..n}\rangle \quad P \vdash \textsf{class } cn\langle f_{1..n}\rangle \textsf{ extends } cn'\langle f_1\ o*\rangle\ ...}{P;\, E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P;\, E \vdash t_1 <: t_2 \quad P;\, E \vdash t_2 <: t_3}{P;\, E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t <: t}$$

$\boxed{P;\, E \vdash \mathit{field}}$

[FIELD]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t\ fd}$$

$\boxed{P \vdash \mathit{field} \in c}$

[FIELD DECLARED]

$$\frac{P \vdash \textsf{class } cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{field}\ ...\}}{P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle \quad P \vdash \textsf{class } cn'\langle g_{1..m}\rangle \textsf{ extends } cn\langle o_{1..n}\rangle ...}{P \vdash \mathit{field}\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\, E \vdash \mathit{method}}$

[METHOD]

$$\frac{E' = E, \textsf{owner } f_{1..n},\ \mathit{constr}*,\ \mathit{arg}* \quad P;\, E' \vdash wf \quad P;\, E' \vdash e : t}{P;\, E \vdash t\ mn\langle f_{1..n}\rangle(\mathit{arg}*) \textsf{ where } \mathit{constr}*\ \{e\}}$$

$\boxed{P \vdash \mathit{meth} \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \textsf{class } cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{meth}\ ...\}}{P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle \quad P \vdash \textsf{class } cn'\langle g_{1..m}\rangle \textsf{ extends } cn\langle o_{1..n}\rangle ...}{P \vdash \mathit{meth}\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\, E \vdash e : t}$

[EXP TYPE]

$$\frac{P;\, E \vdash e : t}{P;\, E \vdash e : t}$$

$\boxed{P;\, E \vdash e : t}$

[EXP SUB]

$$\frac{P;\, E \vdash e : t' \quad P;\, E \vdash t' <: t}{P;\, E \vdash e : t}$$

[EXP REF]

$$\frac{P;\, E \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle}{P;\, E \vdash x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{P;\, E \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad P;\, E \vdash y : t\ [o_1/f_1]..[o_n/f_n]}{P;\, E \vdash x.fd = y : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP NEW]

$$\frac{P;\, E \vdash c}{P;\, E \vdash \textsf{new } c : c}$$

[EXP LET]

$$\frac{\mathit{arg} = t\ x \quad P;\, E \vdash e : t \quad P;\, E, \mathit{arg} \vdash e' : t'}{P;\, E \vdash \textsf{let } (\mathit{arg} = e) \textsf{ in } \{e'\} : t'}$$

[EXP VAR ASSIGN]

$$\frac{P;\, E \vdash x : t \quad P;\, E \vdash e : t}{P;\, E \vdash x = e : t}$$

[EXP VAR]

$$\frac{E = E_1,\ t\ x,\ E_2}{P;\, E \vdash x : t}$$

[EXP INVOKE]

$$\frac{\begin{array}{ll} \multicolumn{2}{c}{P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j \in 1..k}) \textsf{ where } \mathit{constr}*\ ...) \in cn\langle f_{1..n}\rangle} \\ P;\, E \vdash x : cn\langle o_{1..n}\rangle & P;\, E \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\, E \vdash o_i \succeq o_1 & P;\, E \vdash \mathit{constr}\ [o_1/f_1]..[o_m/f_m] \end{array}}{P;\, E \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

Chapter 3

Effects Clauses, Unique Pointers, and Immutable Objects

SafeJava is the first static type system that combines object encapsulation with effects clauses, unique pointers, and immutable objects.

Effects clauses [106] are useful for specifying assumptions that must hold at method boundaries and enable modular reasoning and checking of programs. SafeJava combines object encapsulation with effects clauses to allow programs to precisely specify the side effects of a method in the presence of subtyping and without representation exposure. SafeJava allows effects clauses to use the name of an object to denote all the objects encapsulated within that object. SafeJava first combined object encapsulation with effects clauses in [26] to prevent data races in multithreaded programs. SafeJava also uses effects clauses to statically enforce various other program properties.

Unique pointers [108, 134] are useful to control object aliasing. SafeJava combines ownership types with unique pointers to express constructs that neither ownership types nor unique pointers alone can express, while enforcing object encapsulation. We first combined ownership types with unique pointers in [26] to support ownership transfer. Recent work [44] proposes a more flexible approach that allows a program to specify a unique *external* pointer to an object; there can be other pointers to that object from objects encapsulated within it. We subsequently adopted this approach in SafeJava.

Immutable objects have many advantages. Unlike mutable objects, they can be shared across multiple aliases without complicating the task of understanding and reasoning about correctness of programs. SafeJava is the first system that extends the notion of immutability to object encapsulation. SafeJava statically verifies that if an object is declared to be immutable, then the program does not modify that object or objects encapsulated within that object. SafeJava first combined ownership types with immutable objects in [26] to allow multiple threads to access an immutable object and its encapsulated objects without synchronization and without causing data races.

The rest of this chapter is organized as follows. Section 3.1 presents effects clauses. Section 3.2 presents a formal description of the type system. Sections 3.3 and 3.4 extend the type system to support objects with unique pointers and immutable objects.

$$meth \quad ::= \quad t\ mn\langle formal^*\rangle(arg^*)\ \textsf{reads}\ (owner^*)\ \textsf{writes}\ (owner^*)\ \textsf{where}\ constr^*\ \{e\}$$

Figure 3-1: Grammar Extensions to Support Effects Clauses

```
class IntVector<vOwner> {
  int  elementCount;
  int  init()     writes(this) { elementCount = 0; }
  int  size()     reads (this) { return elementCount; }
  void add(int x) writes(this) { elementCount++;  ... }
}
class IntStack<sOwner> {
  IntVector<this> vec;
  int  init()      writes(this) { vec = new IntVector<this>;  vec.init(); }
  void push(int x) writes(this) { vec.add(x); }
}
void m<sO,vO> (IntStack<sO> s, IntVector<vO> v) writes (s) reads (v) where !(v >= s) !(s >= v) {
  int n = v.size(); s.push(3); assert(n == v.size());
}
```

Figure 3-2: Using Effects Clauses to Enable Modular Reasoning

3.1 Effects Clauses

SafeJava combines encapsulation with effects clauses [106] which are useful for specifying assumptions that must hold at method boundaries and enable modular checking of programs.

Effects clauses have been incorporated into many program formalisms, specification languages, and program checkers. Examples include Morgan's specification statement [110], Z [126], Larch [82], JML [96], and ESC [57, 69]. SafeJava uses effects clauses to statically verify various program properties, as we show in subsequent chapters of this thesis.

Figure 3-1 presents extensions to the grammar shown in Figures 2-5 and 2-6 to support effects clauses. SafeJava allows programmers to specify *reads* and *writes* clauses. Consider a method that specifies that it writes $(w_1, ..., w_n)$ and reads $(r_1, ..., r_m)$. The method can write an object x (or call methods that write x) only if $(w_i \succeq x)$ for some $i \in \{1..n\}$. The method can read an object y (or call methods that read y) only if $(w_i \succeq y)$ or $(r_j \succeq y)$, for some $i \in \{1..n\}$, $j \in \{1..m\}$. SafeJava thus allows a method to both read and write objects named in its writes clause.

In a language that does not have a notion of object encapsulation, it is often difficult to precisely specify the side effects of a method without exposing the representation of its object. Consider, for example, the writes clause on the push method of a Stack object that is implemented using a Vector. Since the push method may write the Vector, the writes clause must specify the name of the Vector, thus exposing the representation of the Stack object.

The problem gets worse if there are several Stack implementations. For example, in an object-oriented language like Java, one can have an abstract Stack class and different subclasses of Stack that use different representations. The subclasses add new fields to Stack, and the push method in a subclass might write the objects pointed to by the new fields. In this case it is impossible to specify the writes clause of the push method of Stack, because the new fields added by subclasses are not visible in the scope of the abstract Stack class.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) writes(this) {
    TNode<this, TOwner> newNode =
      new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() writes(this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  TNode<nodeOwner, TOwner> next;  T<TOwner> value;

  TNode(T<TOwner> v, TNode<nodeOwner, TOwner> n) writes(this) {
    this.value = v;  this.next  = n;
  }
  T<TOwner> value()                reads (this) { return value; }
  TNode<nodeOwner, TOwner> next() writes(this) { return next;  }
}

class T<TOwner> { }
```

Figure 3-3: TStack With Effects Clauses

To solve this problem, one must use some abstraction mechanism with which one can refer to the objects that are not in scope without directly mentioning their names. SafeJava allows effects clauses to use the name of an object o to denote all the objects (reflexively and transitively) owned by o. The push method of Stack can declare that it writes the corresponding Stack object. If a subclass adds a new field f, it can declare that the object pointed to by f is owned by the Stack object. The push method in the subclass is then allowed to write the object pointed to by f.

Figure 3-2 presents an example that shows how ownership types and effects can be used to enable modular reasoning about programs in the presence of subtyping.[1] The example shows an IntStack implemented using an IntVector vec. (We borrowed this example from [99].) The example has a method m that receives two arguments: an IntStack s and an IntVector v. The condition in the assert statement in m can be true only if v is not aliased to s.vec.

In the example, the method m uses a where clause to specify that ($s \not\succeq v$) and ($v \not\succeq s$). Since the ownership relation forms a tree (see Figure 2-4), this constraint implies that v cannot be aliased to s.vec. Furthermore, IntVector.size declares that it only reads objects owned by the IntVector, and IntStack.push declares that it only writes (and reads) objects owned by the IntStack. Therefore, it is possible to reason locally that v.size and s.push cannot interfere, and thus the condition in the assert statement in m must be true.

Figure 3-3 presents another example that shows the effects clauses for a stack of T objects implemented using a linked list. Besides the effects clauses, the code in this example is identical to that in Figure 2-7.

[1] As mentioned before, all the examples in this thesis use a language that is syntactically closer to Java.

Researchers have also proposed data groups [97, 99] that can be used to name groups of objects in an effects clause to write modular specifications in the presence of subtyping. However, unlike the ownership relation in SafeJava that forms a tree (see Figure 2-4), the relation between data groups forms a directed acyclic graph (DAG). Therefore, data groups cannot be used to statically verify the non-interference of two methods. Data groups are implemented using a theorem prover, and in principle, can be very flexible. However, *pivot uniqueness* in [99] imposes drastic restrictions on pivot fields. SafeJava does not impose such restrictions; it only requires that the owner of an object be unique. In [99], the *owner exclusion* constraint is hard coded. In SafeJava, programmers can specify arbitrary constraints on owners using where clauses; owner exclusion can be a default.

3.2 Formal Description

This section presents a formal description of our type system that includes ownership types and effects. It builds on the type system presented in Section 2.3. To simplify the presentation of the key ideas, we exclude inner classes from the formal description. We presented the grammar in Figures 2-5, 2-6, and 3-1. This section describes some of the important rules for type checking. The full set of rules are in Appendix 3.A at the end of this chapter.

The core of our type system is a set of rules for reasoning about the typing judgment: $P;\, E;\, R;\, W \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. R and W must subsume the read and write effects of e. t is the type of e.

We define a typing environment as: $E ::= \emptyset \mid E,\, t\ x \mid E,\, \text{owner}\ f \mid E,\, \mathit{constr}$
The typing environment contains the declared types of variables, the declared owner parameters, and the declared constraints among owners.

We define read and write effects as: $R, W ::= o_{1..n}$
R and W contain the declared read and write effects.

A useful auxiliary judgment is $P;\, E \vdash \mathit{constr}$, where *constr* is an ownership constraint of the form either $o_1 \succeq o_2$ or $o_1 \not\succeq o_2$. The judgment states that the constraint holds in the typing environment E. The rules for this judgment are as follows:

[CONSTR ENV]
$$\frac{E = E_1,\, \mathit{constr},\, E_2}{P;\, E \vdash \mathit{constr}}$$

[OWNER $\succeq$]
$$\frac{P;\, E \vdash e : cn\langle o_{1..n}\rangle}{P;\, E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]
$$\frac{P;\, E \vdash_{\text{owner}} o}{P;\, E \vdash (\mathsf{world} \succeq o)}$$

[REFL $\succeq$]
$$\frac{P;\, E \vdash_{\text{owner}} o}{P;\, E \vdash (o \succeq o)}$$

[TRANS $\succeq$]
$$\frac{\begin{array}{c}P;\, E \vdash (o_3 \succeq o_2)\\ P;\, E \vdash (o_2 \succeq o_1)\end{array}}{P;\, E \vdash (o_3 \succeq o_1)}$$

The first rule states that if a constraint is part of a class or method declaration, then the type checker assumes that constraint within the scope of that class or method. The second rule states that the first owner parameter of a type owns the corresponding object. (Recall this fact from Section 2.2.2). The third rule states that world transitively owns all objects. The fourth and fifth rules state that the ownership relation is reflexive and transitive.

The judgment $P;\, E \vdash X \succeq Y$ states that effects X subsume effects Y. The rule for this checks that for each owner y_j in Y, there is an owner x_i in X such that $x_i \succeq y_j$.

[$X \succeq Y$]
$$\frac{X = x_{1..n} \qquad Y = y_{1..m} \qquad \forall_{j \in \{1..m\}}\ \exists_{i \in \{1..n\}}\ (x_i \succeq y_j)}{P;\, E \vdash (X \succeq Y)}$$

The rule for creating a new object of type $cn\langle o_{1..n}\rangle$ must ensure that the type $cn\langle o_{1..n}\rangle$ is valid. The rule for that checks that $cn\langle f_{1..n}\rangle$ is a valid class; that owners $o_1..o_n$ are valid owners (a valid owner is either this, or world, or a formal owner parameter); that $o_i \succeq o_1$ for all $i \in \{1..n\}$ (recall this constraint from Section 2.2.3); and that all the other constraints in the declaration of class cn are satisfied. Note that since each constraint *constr* is declared inside the class, it might contain occurrences of the formal owner parameters $f_1..f_n$. When *constr* is used outside the class, one must rename the formal parameters with their corresponding actual owner parameters $o_1..o_n$.

[EXP NEW]

$$\frac{P;\, E \vdash cn\langle o_{1..n}\rangle}{P;\, E;\, R;\, W \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle \ldots\ \mathsf{where}\ constr* \ldots \\ P;\, E \vdash_{\mathsf{owner}} o_i \quad P;\, E \vdash o_i \succeq o_1 \quad P;\, E \vdash constr\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E \vdash cn\langle o_{1..n}\rangle}$$

The rules for subtyping are similar to those in parametric polymorphism [3, 29, 32, 90, 112, 132], except that the first owner parameter of the supertype must be the same as that of the subtype. The first owners have to match because they are special, in that they own the corresponding objects. The subtyping relation is reflexive and transitive.

[SUBTYPE C]

$$\frac{\begin{array}{c} P;\, E \vdash cn\langle o_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ cn'\langle f_1\ o*\rangle \ldots \end{array}}{P;\, E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{\begin{array}{c} P;\, E \vdash t_1 <: t_2 \\ P;\, E \vdash t_2 <: t_3 \end{array}}{P;\, E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t <: t}$$

The rule for a let expression simply adds the new variable and its type to the type environment. The rules for reading and writing a variable look up the type of the variable from the type environment.

[EXP LET]

$$\frac{\begin{array}{c} arg = t\ x \quad P;\, E;\, R;\, W \vdash e : t \\ P;\, E,\, arg;\, R;\, W \vdash e' : t' \end{array}}{P;\, E;\, R;\, W \vdash \mathsf{let}\ (arg = e)\ \mathsf{in}\ \{e'\} : t'}$$

[EXP VAR]

$$\frac{E = E_1,\, t\ x,\, E_2}{P;\, E;\, R;\, W \vdash x : t}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P;\, E;\, R;\, W \vdash x : t \\ P;\, E;\, R;\, W \vdash e : t \end{array}}{P;\, E;\, R;\, W \vdash x = e : t}$$

The rule for accessing field $x.fd$ checks that x is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t. It also verifies that x is included in the permitted read effects R. Note that the rule renames t when it used outside class cn.

[EXP REF]

$$\frac{P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad R = R_1,\, r,\, R_2 \quad r \succeq x}{P;\, E;\, R;\, W \vdash x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

The rule for assigning to a field is similar.

[EXP REF ASSIGN]

$$\frac{\begin{array}{c} P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad W = W_1,\, w,\, W_2 \quad w \succeq x \\ P;\, E;\, R;\, W \vdash y : t\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E;\, R;\, W \vdash x.fd = y : t\ [o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method on an object checks that the type of that object declares or inherits the method, and that the method arguments are of the right type. Since methods in our system may be parameterized, the rule checks that the method parameters are instantiated with valid owners, and that the constraints on the owners satisfied. The rule also checks that the permitted read and write effects R and W subsume the effects mentioned in the reads and writes clauses of the method. The rule appropriately renames types and effects when they are used outside their declared context.

[EXP INVOKE]

$$\frac{\begin{array}{c} P \vdash (t\ mn\langle f_{(n+1)..m}\rangle (t_j\ y_j\ ^{j\in 1..k})\ \mathsf{reads}(r_{1..r})\ \mathsf{writes}(w_{1..w})\ \mathsf{where}\ constr*\ \{e\}) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ R;\ W \vdash x : cn\langle o_{1..n}\rangle \quad P;\ E \vdash o_i \succeq o_1 \quad P;\ E \vdash constr\ [o_1/f_1]..[o_m/f_m] \\ P;\ E;\ R;\ W \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\ E \vdash R \succeq r_{1..r}\ [o_1/f_1]..[o_m/f_m] \quad P;\ E \vdash W \succeq w_{1..w}\ [o_1/f_1]..[o_m/f_m] \end{array}}{P;\ E;\ R;\ W \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

3.3 Unique Pointers

This section presents extensions to our type system to support unique pointers [108, 134]. Unique pointers are useful to control object aliasing. Unique pointers have been used in low level languages to support safe explicit memory deallocation [48, 84] and to track resource usage [52, 53]. Ownership types combined with unique pointers can express constructs that neither ownership types nor unique pointers alone can express, while enforcing object encapsulation. We first combined ownership types with unique pointers in [26] to support ownership transfer. Recent work [44] proposes a more flexible approach that allows a program to specify a unique *external* pointer to an object; there can be other pointers to that object from objects encapsulated within it. We subsequently adopted this approach.

SafeJava statically enforces the uniqueness property shown in Figure 3-5. Property SJ6 states that if a method-local variable or an object field x is declared to be the unique pointer to an object o, then there can be no other pointer to o, except from fields of objects encapsulated within o. In other words, x is the unique *external* pointer to o.

Creating Unique Pointers

Figure 3-4 presents grammar extensions to support unique pointers. The instruction new creates a unique pointer. A variable (or field) x can be declared to be a unique pointer by instantiating its type with unique:o as the first parameter. o is an owner. If o is world, then programmers can simply use unique. The owner o restricts the set of variables that x can be transferred to. In particular, x can only be transferred to variables where o is in scope. This restriction is necessary to enforce object encapsulation in the presence of unique pointers.

Figure 3-6 presents some examples, which shows client code for the TStack in Figure 2-7. In the figure, TStack⟨unique, world⟩ u3 is equivalent to TStack⟨unique:world, world⟩ u3 and it declares that u3 is a unique pointer to a TStack object. TStack⟨unique:this, this⟩ u1 declares that u1 is a unique pointer to a TStack object, and that u1 can only be transferred to variables where this is in scope. That is, u1 cannot be transferred outside the encapsulation boundary of the TStackClient object.

$$
\begin{array}{rcl}
o_u & ::= & \textsf{unique} \mid \textsf{unique} : owner \\
c & ::= & ... \mid cn\langle o_u\ \ owner{*}\rangle \mid \textsf{Object}\langle o_u\ \ owner{*}\rangle \mid c.cn\langle o_u\ \ owner{*}\rangle \\
e & ::= & ... \mid x\text{--} \mid x.fd\text{--} \mid x.fd = y\text{--} \mid \textsf{borrow}\ (t\ x = y)\ \{\ e\ \}
\end{array}
$$

Figure 3-4: Grammar Extensions to Support Unique Pointers

SJ6. If a method-local variable or an object field x is declared to be the unique pointer to an object o, then:

i. No other method-local variable can point to o.

ii. If another field x' of an object o' points to o then $(o \succeq o')$.

SJ7. If a method-local variable or an object field x is declared to point to an immutable object o, then the program cannot write to o or any object o' where $(o \succeq o')$.

Figure 3-5: Properties of Objects With Unique Pointers and Immutable Objects

Recall from Section 2.2.3 that to enforce object encapsulation, SafeJava requires that in every type $cn\langle o_1, ..., o_n\rangle$ with multiple owners, $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. For a type $cn\langle \textsf{unique}{:}o_1\ \ o_2, ..., o_n\rangle$, the restriction is that $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. This restriction is necessary to enforce object encapsulation in the presence of unique variables. Thus, the type of TStack u4 in Figure 2-7 is illegal because $(\textsf{this} \not\succeq \textsf{world})$.

The rules for creating a unique pointer of a valid type are shown below. These rules are similar to those in Section 2.3.

[EXP NEW UNIQUE]

$$
\frac{P;\, E \vdash cn\langle \textsf{unique}{:}o_1\ \ o_{2..n}\rangle}{P;\, E;\, R;\, W \vdash \textsf{new}\ cn\langle \textsf{unique}{:}o_1\ \ o_{2..n}\rangle\ :\ cn\langle \textsf{unique}{:}o_1\ \ o_{2..n}\rangle}
$$

[TYPE C UNIQUE]

$$
\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle ...\ \textsf{where}\ constr{*}\ ... \quad P;\, E \vdash_{\text{owner}} o_i \quad P;\, E \vdash o_i \succeq o_1 \quad P;\, E \vdash constr\ [o_1/f_1]..[o_n/f_n]}{P;\, E \vdash cn\langle \textsf{unique}{:}o_1\ \ o_{2..n}\rangle}
$$

Transferring Unique Pointers

Regular variables cannot be assigned to unique variables. Pointers in unique variables can be transferred to other regular or unique variables only by using the following syntax. x and y are assumed to be unique variables in the example below.

```
y = x--;  // y = x; x = null;
m(y--);   // m(y);  y = null;
```

Guava [11] uses a similar syntax to transfer objects between variables. This syntax is inspired by the following C expression syntax. i and j are assumed to be integer variables below.

```
j = i--;  // j = i; i = i-1;
m(j--);   // m(j);  j = j-1;
```

In the above example, if the x is a method-local variable and is not subsequently used within the method, an optimizing compiler will eliminate the instruction x=null as dead

```
class TStackClient<clientOwner> {
  void test() writes(world) {
    TStack<this,  this>  s1;
    TStack<this,  world> s2;
    TStack<world, world> s3;
 /* TStack<world, this>  s4; illegal! */

    TStack<unique:this, this>  u1, v1;
    TStack<unique:this, world> u2, v2;
    TStack<unique,      world> u3, v3;     // Equivalent to:  TStack<unique:world, world> u3, v3;
 /* TStack<unique,      this>  u4, v4; illegal! */

    v1 = u1--;  v2 = u2--;  v3 = u3--;        v2 = u3--;
    s1 = u1--;  s2 = u2--;  s3 = u3--;        s2 = u3--;

    borrow (TStack<f, this> x1 = u1) {     // x1 = u1--;
      x1.push(...);
      x1.pop(); }
      TStack<f, this> x2 = x1;
      x2.push(...);
      x1.pop();
    }                                      // u1 = x1--;
}}
```

Figure 3-6: TStacks With Unique Pointers

code. By using the x-- construct, we are in effect shifting some checking from compile time to runtime. If x is subsequently dereferenced before being reinitialized, the system will raise a NullPointer exception at runtime.

A unique type with first parameter unique:o is a subtype of another unique type with first parameter unique:o' if $(o \succeq o')$. A variable with a unique type t can be transferred to another variable with a unique type t' only if t is a subtype of t'. Thus, in Figure 3-6, u3 can be transferred to u2, v2, and v3. Similarly, a unique type with first parameter unique:o is a subtype of a regular type with first parameter o' if $(o \succeq o')$. A variable with a unique type t can be transferred to a variable with a regular type t' only if t is a subtype of t'. Thus, in Figure 3-6, u3 can be transferred to s2 and s3.

The rules for subtyping with unique pointers are shown below. These rules are similar to the rules for subtyping in Section 2.3.

[SUBTYPE C UNIQUE 1]

$$\frac{P;\, E \vdash cn\langle \textsf{unique:}o\ o_{2..n}\rangle \qquad P;\, E \vdash o \succeq o'}{P;\, E \vdash cn\langle \textsf{unique:}o\ o_{2..n}\rangle <: cn\langle \textsf{unique:}o'\ o_{2..n}\rangle}$$

[SUBTYPE C UNIQUE 2]

$$\frac{P;\, E \vdash cn\langle \textsf{unique:}o_1\ o_{2..n}\rangle \qquad P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \textsf{extends}\ cn'\langle f_1\ o*\rangle\ ...}{P;\, E \vdash cn\langle \textsf{unique:}o_1\ o_{2..n}\rangle <: cn\langle \textsf{unique:}f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

Because SafeJava combines ownership types and unique pointers in a common framework, it supports constructs that neither ownership types nor unique pointers alone can support, while enforcing object encapsulation. Figure 3-7 provides an illustration. The example is adopted from a *stock quote* server we had implemented in [26]. In the example, StockQuoteHandler is initialized with an externally created Socket object, which is subsequently encapsulated within the StockQuoteHandler.

```
class StockQuoteHandler ... {
  Socket<this> s;
  StockQuoteHandler(Socket<unique> s) ... {
    this.s = s--;  // this.s = s; s = null;
  } ...
}
class Main ... {
  void serveQuotes(...) {
    Socket<unique> s = ...;
    StockQuoteHandler h = new StockQuoteHandler(s--);
    ...
}}
```

Figure 3-7: Using Ownership Types and Unique Pointers to Enforce Encapsulation

Using Unique Pointers

Programmers often write code that creates temporary aliases to objects, e.g., by storing pointers in loop variables or passing them to functions. To allow temporarily aliases to objects with unique pointers, SafeJava allows a regular variable to temporarily borrow the pointer in a unique variable [44]. In Figure 3-6, x1 borrows u1. This operation introduces a fresh owner f where $(\mathsf{this} \succeq \mathsf{f})$, introduces a fresh variable x1 owned by f, and transfers u1 to x1. Within the scope of the borrow, f owns x1 and the program can freely create aliases to x1. Note that u1 is unusable within the scope of borrow; attempts to access u1 would raise a NullPointer exception at runtime. At the end of the scope of borrow, the program transfers x1 back to u1. All the other aliases to x1 are no longer usable because their owner f is no longer in scope. u1 thus becomes the unique *external* pointer to that object. Note that there can be other pointers to u1 from objects encapsulated within u1.

The static type checking rule for borrow is shown below. [44] contains a proof that borrow preserves the external uniqueness property shown in Figure 3-5.

[EXP BORROW]

$$\frac{P;\ E;\ R;\ W \vdash y : cn\langle \mathsf{unique}{:}o\ o_{2..n}\rangle \qquad P;\ E,\ \mathsf{owner}\ f,\ (o \succeq f),\ cn\langle f\ o_{2..n}\rangle\ x;\ R;\ W \vdash e : t}{P;\ E;\ R;\ W \vdash \mathsf{borrow}\ (cn\langle f\ o_{2..n}\rangle\ x = y)\ \mathsf{in}\ \{e\} : t}$$

3.4 Immutable Objects

Immutable objects have many advantages. Unlike mutable objects, they can be shared across multiple aliases without complicating the task of understanding and reasoning about correctness of programs. SafeJava is the first system that extends the notion of immutability to object encapsulation. SafeJava statically enforces the immutability property shown in Figure 3-5. Property SJ7 states that if an object is declared to be immutable, then the program does not modify that object or objects encapsulated within that object. We first combined ownership types with immutable objects in [26] to allow multiple threads to access an immutable object and its encapsulated objects without synchronization and without causing data races.

A variable (or field) x can be declared to be a pointer to an immutable object by instantiating its type with immutable:o as the first parameter. o is an owner. If o is world, then

$$o_{\mathrm{m}} ::= \textsf{immutable} \mid \textsf{immutable} : owner$$
$$c ::= \ldots \mid cn\langle o_{\mathrm{m}}\ owner{*}\rangle \mid \textsf{Object}\langle o_{\mathrm{m}}\ owner{*}\rangle \mid c.cn\langle o_{\mathrm{m}}\ owner{*}\rangle$$

Figure 3-8: Grammar Extensions to Support Immutable Objects

```
void m<sO,vO> (IntStack<sO> s, IntVector<vO> v) writes (s) reads (v) where !(v >= s) !(s >= v) {
  int n = v.size(); s.push(3); assert(n == v.size());
}

IntVector<unique>    v1 = new IntVector<unique>;  borrow (IntVector<o> v2 = v1) { v1.init(); }
IntVector<immutable> v  = v1--;
IntStack<world>      s  = new IntStack<world>;    s.init();

m(s, v);
```

Figure 3-9: Using Immutable Objects

programmers can simply use immutable. For a type $cn\langle \textsf{immutable:}o_1\ \ o_2, ..., o_n\rangle$, SafeJava statically checks that $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. This restriction is necessary to enforce object encapsulation in the presence of immutable objects.

The type checking rule for a valid immutable type is shown below.

[TYPE C IMMUTABLE]

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle \ldots\ \textsf{where}\ constr{*} \ldots \qquad P; E \vdash_{\mathrm{owner}} o_i \qquad P; E \vdash o_i \succeq o_1 \qquad P; E \vdash constr\ [o_1/f_1]..[o_n/f_n]}{P; E \vdash cn\langle \textsf{immutable:}o_1\ o_{2..n}\rangle}$$

Figure 3-9 presents an example, that contains client code for IntStack and IntVector shown in Figure 3-2. In the figure, the program first creates and initializes a IntVector object that has a unique reference. The program then transfers the reference to a variable of immutable type. In general, this is the mechanism SafeJava uses to support the initialization of immutable objects.

The rule for subtyping with immutable objects are shown below.

[SUBTYPE C IMMUTABLE 1]

$$\frac{P; E \vdash cn\langle \textsf{unique:}o_1\ o_{2..n}\rangle}{P; E \vdash cn\langle \textsf{unique:}o_1\ o_{2..n}\rangle <: cn\langle \textsf{immutable:}o_1\ o_{2..n}\rangle}$$

[SUBTYPE C IMMUTABLE 2]

$$\frac{P; E \vdash cn\langle \textsf{immutable:}o\ o_{2..n}\rangle \qquad P; E \vdash o \succeq o'}{P; E \vdash cn\langle \textsf{immutable:}o\ o_{2..n}\rangle <: cn\langle \textsf{immutable:}o'\ o_{2..n}\rangle}$$

[SUBTYPE C IMMUTABLE 3]

$$\frac{P; E \vdash cn\langle \textsf{immutable:}o_1\ o_{2..n}\rangle \qquad P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \textsf{extends}\ cn'\langle f_1\ o{*}\rangle \ldots}{P; E \vdash cn\langle \textsf{immutable:}o_1\ o_{2..n}\rangle <: cn\langle \textsf{immutable:}f_1\ o{*}\rangle\ [o_1/f_1]..[o_n/f_n]}$$

A program cannot write to an immutable object or an encapsulated subobject of an immutable object. A program can pass an immutable object o as an argument to a method only if it is not part of the write effects W of the method, that is, only if ($o \not\succeq W$) and ($W \not\succeq o$). In Figure 3-9, method m declares that it only writes s, and that ($\mathsf{s} \not\succeq \mathsf{v}$) and ($\mathsf{v} \not\succeq \mathsf{s}$). Therefore, it is safe to pass the immutable v to m.

The rule for writing to an object field in the presence of immutable objects checks that the program does not write any immutable objects. The rest of the rule is identical to that in Section 2.3.

[EXP REF ASSIGN]

$$\begin{array}{c}\text{Immutable}(z,\, E) \quad \stackrel{\text{def}}{=} \quad E = E_1,\, cn'\langle \mathsf{immutable}{:}o'_1\ o'_{2..n}\rangle\ z,\, E_2 \\ \forall_z\ \text{Immutable}(z,\, E) \quad \Longrightarrow \quad (P;\, E \vdash z \not\succeq x)\end{array}$$

$$\frac{\begin{array}{c}P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \qquad W = W_1,\, w,\, W_2 \qquad w \succeq x \\ P;\, E;\, R;\, W \vdash y : t\ [o_1/f_1]..[o_n/f_n]\end{array}}{P;\, E;\, R;\, W \vdash x.fd = y : t\ [o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method in the presence of immutable objects checks that the method does not write any immutable objects. The rest of the rule is identical to that in Section 2.3.

[EXP INVOKE IMMUTABLE]

$$\begin{array}{c}\text{Immutable}(z,\, E) \quad \stackrel{\text{def}}{=} \quad E = E_1,\, cn'\langle \mathsf{immutable}{:}o'_1\ o'_{2..n}\rangle\ z,\, E_2 \\ w'_i = w_i\ [o_1/f_1]..[o_m/f_m] \\ \forall_z\ \text{Immutable}(z,\, E) \quad \Longrightarrow \quad \forall_i\ (P;\, E \vdash z \not\succeq w'_i) \wedge (P;\, E \vdash w'_i \not\succeq z)\end{array}$$

$$\frac{\begin{array}{c}P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{reads}(r_{1..r})\ \mathsf{writes}(w_{1..w})\ \mathsf{where}\ constr{*}\ \{e\}) \\ P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \qquad P;\, E;\, R;\, W \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\, E \vdash_{\text{owner}} o_i \qquad P;\, E \vdash o_i \succeq o_1 \qquad P;\, E \vdash constr\ [o_1/f_1]..[o_m/f_m] \\ P;\, E \vdash R \succeq r_{1..r}\ [o_1/f_1]..[o_m/f_m] \qquad P;\, E \vdash W \succeq w_{1..w}\ [o_1/f_1]..[o_m/f_m]\end{array}}{P;\, E;\, R;\, W \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

Conclusions

This chapter presented extensions to the SafeJava type system that combine object encapsulation with effects clauses, unique pointers, and immutable objects. The subsequent chapters of the thesis build on this type system to statically prevent several kinds of programming errors.

3.A Rules for Type Checking

This section formally presents the SafeJava type system that includes ownership types and effects clauses. It builds on the type system presented in Appendix 2.A at the end of Chapter 2.

The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & defn^*\ e \\
defn & ::= & \textsf{class}\ cn\langle formal+\rangle\ \textsf{extends}\ c\ \textsf{where}\ constr^*\ \{field^*\ meth^*\} \\
c & ::= & cn\langle owner+\rangle \mid \textsf{Object}\langle owner\rangle \\
owner & ::= & formal \mid \textsf{world} \mid \textsf{this} \\
constr & ::= & (owner \succeq owner) \mid (owner \not\succeq owner) \\
meth & ::= & t\ mn\langle formal^*\rangle(arg^*)\ \textsf{reads}\ (owner^*)\ \textsf{writes}\ (owner^*)\ \textsf{where}\ constr^*\ \{e\} \\
field & ::= & t\ fd \\
arg & ::= & t\ x \\
t & ::= & c \mid \textsf{int} \\
formal & ::= & f \\
 & & \\
e & ::= & \textsf{new}\ c \mid x \mid x = e \mid \textsf{let}\ (arg{=}e)\ \textsf{in}\ \{e\} \mid x.fd \mid x.fd = y \mid x.mn\langle owner^*\rangle(y^*) \\
 & & \\
cn & \in & \text{class names} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
x,y & \in & \text{variable names} \\
f & \in & \text{owner names}
\end{array}
$$

We first define a number of predicates used in the type system. These predicates are based on similar predicates from [70]. We refer the reader to that paper for their precise formulation.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
FieldsOnce(P)	No class contains two fields, declared or inherited, with same name
MethodsOncePerClass(P)	No class contains two methods with same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden. The read and write effects of an overriding method must be superseded by those of the overridden methods

The core of our type system is a set of rules for reasoning about the typing judgment: $P;\ E;\ R;\ W \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. R and W must subsume the read and write effects of e. t is the type of e.

We define a typing environment as $E ::= \emptyset \mid E, t\ x \mid E, \text{owner}\ f \mid E, constr$
The typing environment contains the declared types of variables, the declared owner parameters, and the declared constraints among owners.

We define read and write effects as $R, W ::= o_{1..n}$
R and W contain the declared read and write effects.

We define the type system using the following judgments. We present the typing rules for these judgments after that.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn}$	defn is a well-formed class
$P; E \vdash_{\text{owner}} o$	o is an owner
$P; E \vdash \mathit{constr}$	constraint constr is satisfied
$P; E \vdash X \succeq Y$	effect X subsumes effect Y
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash wf$	typing environment E is well-formed
$P \vdash \mathit{field} \in c$	class c declares/inherits field
$P \vdash \mathit{meth} \in c$	class c declares/inherits meth
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash e : t$	expression e has type t
$P; E; R; W \vdash e : t$	expression e has type t and its read/write effects are subsumed by R/W

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c}\mathit{WFClasses(P)} \quad \mathit{ClassOnce(P)} \quad \mathit{FieldsOnce(P)} \\ \mathit{MethodsOncePerClass(P)} \quad \mathit{OverridesOK(P)} \\ P = \mathit{defn}_{1..n}\ e \quad P \vdash \mathit{defn}_i \quad P; \emptyset; \mathsf{world}; \mathsf{world} \vdash e : t\end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]

$$\frac{\begin{array}{c}E = cn\langle f_{1..n}\rangle\ \mathsf{this},\ \mathsf{owner}\ f_{1..n},\ f_i \succeq f_1,\ \mathit{constr}* \\ P; E \vdash wf \qquad P; E \vdash c' \\ P; E \vdash \mathit{field}_i \qquad P; E \vdash \mathit{meth}_i\end{array}}{\begin{array}{r}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c'\ \mathsf{where}\ \mathit{constr}* \\ \{\mathit{field}*\ \mathit{meth}*\}\end{array}}$$

$\boxed{P; E \vdash \mathit{constr}}$

[CONSTR ENV]

$$\frac{E = E_1,\ \mathit{constr},\ E_2}{P; E \vdash \mathit{constr}}$$

[OWNER $\succeq$]

$$\frac{P; E \vdash e : cn\langle o_{1..n}\rangle}{P; E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (\mathsf{world} \succeq o)}$$

[REFL $\succeq$]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (o \succeq o)}$$

[TRANS $\succeq$]

$$\frac{\begin{array}{c}P; E \vdash (o_3 \succeq o_2) \\ P; E \vdash (o_2 \succeq o_1)\end{array}}{P; E \vdash (o_3 \succeq o_1)}$$

$\boxed{P; E \vdash X \succeq Y}$

[$X \succeq Y$]

$$\frac{\begin{array}{c}X = x_{1..n} \quad Y = y_{1..m} \\ \forall_{j \in \{1..m\}}\ \exists_{i \in \{1..n\}}\ (P; E \vdash x_i \succeq y_j)\end{array}}{P; E \vdash (X \succeq Y)}$$

$\boxed{P; E \vdash_{\text{owner}} o}$

[OWNER WORLD]

$$\frac{}{P; E \vdash_{\text{owner}} \mathsf{world}}$$

[OWNER FORMAL]

$$\frac{E = E_1,\ \mathsf{owner}\ f,\ E_2}{P; E \vdash_{\text{owner}} f}$$

[OWNER THIS]

$$\frac{E = E_1,\ c\ \mathsf{this},\ E_2}{P; E \vdash_{\text{owner}} \mathsf{this}}$$

$\boxed{P; E \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P; \emptyset \vdash wf}$$

[ENV X]

$$\frac{\begin{array}{c}P; E \vdash t \\ x \notin \mathrm{Dom}(E) \\ P; E \vdash wf\end{array}}{P; E, t\ x \vdash wf}$$

[ENV OWNER]

$$\frac{\begin{array}{c}f \notin \mathrm{Dom}(E) \\ P; E \vdash wf\end{array}}{P; E,\ \mathsf{owner}\ f \vdash wf}$$

[ENV CONSTR]

$$\frac{\begin{array}{c}\mathit{constr} = (o' \succeq o) \vee \mathit{constr} = (o' \not\succeq o) \\ P; E \vdash wf \qquad P; E \vdash_{\text{owner}} o, o' \\ E' = E,\ \mathit{constr} \\ \nexists_{x,y}\ (P; E' \vdash y \succeq x) \wedge (P; E' \vdash y \not\succeq x)\end{array}}{P; E,\ \mathit{constr} \vdash wf}$$

$\boxed{P; E \vdash t}$

[TYPE INT]

$$\frac{}{P; E \vdash \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle ...\ \mathsf{where}\ \mathit{constr}*\ ... \\ P; E \vdash_{\text{owner}} o_i \quad P; E \vdash o_i \succeq o_1 \quad P; E \vdash \mathit{constr}\ [o_1/f_1]..[o_n/f_n]\end{array}}{P; E \vdash cn\langle o_{1..n}\rangle}$$

$\boxed{P;\, E \vdash t_1 <: t_2}$

[SUBTYPE C]

$$\frac{P;\, E \vdash cn\langle o_{1..n}\rangle \quad P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \textsf{extends}\ cn'\langle f_1\ o*\rangle\ ...}{P;\, E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P;\, E \vdash t_1 <: t_2 \quad P;\, E \vdash t_2 <: t_3}{P;\, E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t <: t}$$

$\boxed{P;\, E \vdash \mathit{field}}$

[FIELD]

$$\frac{P;\, E \vdash t}{P;\, E \vdash t\ fd}$$

$\boxed{P \vdash \mathit{field} \in c}$

[FIELD DECLARED]

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{field}\ ...\}}{P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle \quad P \vdash \textsf{class}\ cn'\langle g_{1..m}\rangle\ \textsf{extends}\ cn\langle o_{1..n}\rangle ...}{P \vdash \mathit{field}\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\, E \vdash \mathit{method}}$

[METHOD]

$$\frac{E' = E,\ \textsf{owner}\ f_{1..n},\ constr*,\ arg* \quad P;\, E' \vdash wf \quad P;\, E';\, r*, w*;\, w* \vdash e : t}{P;\, E \vdash t\ mn\langle f_{1..n}\rangle(arg*)\ \textsf{reads}(r*)\ \textsf{writes}(w*)\ \textsf{where}\ constr*\ \{e\}}$$

$\boxed{P \vdash \mathit{meth} \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle ...\ \{...\ \mathit{meth}\ ...\}}{P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle \quad P \vdash \textsf{class}\ cn'\langle g_{1..m}\rangle\ \textsf{extends}\ cn\langle o_{1..n}\rangle ...}{P \vdash \mathit{meth}\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\, E \vdash e : t}$

[EXP TYPE]

$$\frac{P;\, E;\, \textsf{world};\, \textsf{world} \vdash e : t}{P;\, E \vdash e : t}$$

$\boxed{P;\, E;\, R;\, W \vdash e : t}$

[EXP SUB]

$$\frac{P;\, E;\, R;\, W \vdash e : t' \quad P;\, E;\, R;\, W \vdash t' <: t}{P;\, E;\, R;\, W \vdash e : t}$$

[EXP REF]

$$\frac{P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad R = R_1,\, r,\, R_2 \quad r \succeq x}{P;\, E;\, R;\, W \vdash x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad W = W_1,\, w,\, W_2 \quad w \succeq x \quad P;\, E;\, R;\, W \vdash y : t\ [o_1/f_1]..[o_n/f_n]}{P;\, E;\, R;\, W \vdash x.fd = y : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP NEW]

$$\frac{P;\, E \vdash c}{P;\, E;\, R;\, W \vdash \textsf{new}\ c : c}$$

[EXP LET]

$$\frac{arg = t\ x \quad P;\, E;\, R;\, W \vdash e : t \quad P;\, E,\, arg;\, R;\, W \vdash e' : t'}{P;\, E;\, R;\, W \vdash \textsf{let}\ (arg = e)\ \textsf{in}\ \{e'\} : t'}$$

[EXP VAR ASSIGN]

$$\frac{P;\, E;\, R;\, W \vdash x : t \quad P;\, E;\, R;\, W \vdash e : t}{P;\, E;\, R;\, W \vdash x = e : t}$$

[EXP VAR]

$$\frac{E = E_1,\, t\ x,\, E_2}{P;\, E;\, R;\, W \vdash x : t}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \textsf{reads}(r_{1..r})\ \textsf{writes}(w_{1..w})\ \textsf{where}\ constr*\ ...) \in cn\langle f_{1..n}\rangle \\ \begin{array}{ll} P;\, E;\, R;\, W \vdash x : cn\langle o_{1..n}\rangle & P;\, E;\, R;\, W \vdash x_j : t_j\ [o_1/f_1]..[o_m/f_m] \\ P;\, E \vdash o_i \succeq o_1 & P;\, E \vdash R \succeq r_{1..r}\ [o_1/f_1]..[o_m/f_m] \\ P;\, E \vdash constr\ [o_1/f_1]..[o_m/f_m] & P;\, E \vdash W \succeq w_{1..w}\ [o_1/f_1]..[o_m/f_m] \end{array} \end{array}}{P;\, E;\, R;\, W \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\ [o_1/f_1]..[o_m/f_m]}$$

Chapter 4

Preventing Data Races and Deadlocks

Multithreaded programming is becoming a mainstream programming practice. But multithreaded programming is difficult and error prone. Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. A deadlock occurs when there is a set of threads such that every thread in the set is waiting on a lock held by another thread in the set. Synchronization errors in multithreaded programs are timing-dependent, non-deterministic bugs, and are among the most difficult programming errors to detect, reproduce, and eliminate.

SafeJava provides a new static type system for multithreaded programs; well-typed programs in SafeJava are guaranteed to be free of data races and deadlocks. The basic idea is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. SafeJava allow programmers to specify this locking discipline in their programs in the form of type declarations. SafeJava statically verifies that a program is consistent with its type declarations.

The SafeJava type system combines object encapsulation with safe multithreading. Object encapsulation is useful for safe multithreading because the same lock that protects an object can also protect the objects encapsulated within that object.

Preventing Data Races

To prevent data races, programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. The protection mechanism of an object can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either 1) the object is immutable, 2) the object is accessible to a single thread, or 3) there is a unique pointer to the object. Unique pointers are useful to support object migration between threads. The SafeJava type checker statically verifies that a program uses objects only in accordance with their declared protection mechanisms.

The SafeJava type system is significantly more expressive than previously proposed type systems for preventing data races [68, 11]. In particular, SafeJava lets programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. SafeJava does this by introducing a way of parameterizing classes that lets programmers defer the protection mechanism decision from the time when a class is defined to the times when objects of that class are created. Without this flexibility, programmers often must either write a program that acquires redundant locks just to satisfy the type checker, or unnecessarily duplicate code to produce multiple versions of the same classes; these versions differ only in the code that implements the protection mechanisms.

One of the challenges in designing an effective type system is to make it powerful enough to express common programming paradigms. One trivial way to guarantee race-free programs, for example, is to require every thread to acquire the lock on every object before accessing the object. But that would introduce unnecessary synchronization overhead because programmers often know from the logic of their programs that acquiring certain locks is unnecessary.

SafeJava is expressive enough to verify the absence of races in many common situations where a thread accesses an object without acquiring the lock on that object. In particular, it accommodates the following cases:

- **Thread-local objects:** If an object is accessed by only one thread, it needs no synchronization.

  Consider, for example, a Vector class. In SafeJava, programmers can write a generic Vector implementation. A program can then create some thread-local Vector objects—these objects can be accessed without any synchronization. A program can also create Vector objects that are shared between multiple threads—these objects contain their own locks that a thread must acquire before it accesses the objects. Moreover, a program can also create thread-local Vector objects containing only thread-local elements, and thread-local Vector objects containing shared elements, all from the same generic Vector implementation.

  With previous systems, the only way to do this is to have different versions of the Vector class, one for each case. These versions contain the exact same code except for synchronization operations.

- **Objects contained within other objects:** Sometimes, an object is contained within an enclosing data structure. In such cases, it might be redundant to acquire the lock on that object since the same lock that protects the enclosing data structure also protects that object.

  Consider, for example, a Stack implementation that internally uses a Vector. In SafeJava, a program can create a Vector object from the same generic Vector implementation such that the Vector object inherits the protection mechanism of the enclosing Stack object. If the program then creates a Stack object that is thread-local, no synchronization operations is necessary to access the Stack or the Vector. If the program creates a shared Stack object, the same lock that protects the Stack also protects the Vector.

  Previous systems needed multiple Vector and Stack implementations to support these different cases.

- **Objects migrating between threads:** Some programs use serially-shared objects that migrate from one thread to another. Although these objects are shared by multiple threads, they are accessed only by a single thread at a time. Operations on these objects can therefore execute without synchronization. SafeJava uses the notion of unique pointers [108] to support this kind of sharing.

  SafeJava also allows programs to build collection classes that contain unique objects. For example, programmers can create a Queue of unique objects. This is useful to support the producer-consumer paradigm where producer threads insert items into the Queue and consumer threads extract them from the Queue.

- **Immutable objects:** Programs often use immutable objects that are initialized once by a single thread, then read by multiple threads. Because none of the parallel threads writes a immutable object after it is initialized, they can all access the object concurrently without synchronization and without causing data races. SafeJava supports this sharing pattern.

Because the SafeJava type system is expressive and yet guarantees race-freedom, programmers can apply efficient protection mechanisms without risking synchronization errors. For example, the Java libraries contain two different classes to implement resizable arrays: the Vector class and the ArrayList class. The methods of Vector are synchronized, therefore, multiple threads can use Vector objects without creating data races. But Vectors always incur a synchronization overhead, even when used in contexts where synchronization is unnecessary. On the other hand, the methods of ArrayList are not synchronized, therefore, ArrayLists do not incur any unnecessary synchronization overhead. But programs that use ArrayLists risk data races because there is no mechanism in Java to ensure that programs access ArrayLists with appropriate synchronization when they use ArrayLists in multithreaded contexts. SafeJava enables programmers to implement a single generic resizable array class. If a program creates a resizable array object to be concurrently shared between threads, SafeJava ensures that accesses to the array are synchronized. If an array is not concurrently shared, SafeJava allows the program to access the array without synchronization.

Finally, we note in passing that any type system that guarantees race freedom also eliminates issues associated with the use of weak memory consistency models [117]. A detailed explanation of this issue can be found in [11].

Preventing Deadlocks

To prevent deadlocks, programmers partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The SafeJava type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order of lock levels.

The SafeJava type system also provides the following features:

- **Lock Level Polymorphism:** SafeJava allows programmers write code that is polymorphic in lock levels. It also allows programmers to specify a partial order among formal lock level parameters using where clauses [50, 112]. This enables programmers to write code in which the exact levels of some locks are not known statically, but only some ordering constraints among the unknown lock levels are known statically.

- **Support for Condition Variables:** In addition to mutual exclusion locks, SafeJava prevents deadlocks in the presence of condition variables. SafeJava statically enforces the constraint that a thread can invoke e.wait only if the thread holds no locks other than the lock on e. Since a thread releases the lock on e on executing e.wait, the above constraint implies that any thread that is waiting on a condition variable holds no locks. This in turn implies that there cannot be a deadlock that involves a condition variable. SafeJava thus prevents the nested monitor problem [105].

- **Partial-Orders Based on Mutable Trees:** SafeJava also allows programmers to use recursive tree-based data structures to further order the locks within a given lock level. For example, programmers can specify that nodes in a tree must be locked in the *tree order*. SafeJava allows mutations to the data structure that change the partial order at runtime. The SafeJava type checker uses an intraprocedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime.

- **Partial-Orders Based on Monotonic DAGs:** SafeJava also allows programmers to use recursive DAG-based data structures to order the locks within a given lock level. DAG edges cannot be modified once initialized. Only newly created nodes may be added to a DAG by initializing the newly created nodes to contain DAG edges to existing DAG nodes.

- **Runtime Ordering of Locks:** SafeJava supports imposing an arbitrary linear order at runtime on locks within a given lock level. SafeJava also provides a primitive to acquire such locks in the linear order.

Outline

This chapter extends the type system we presented in Chapter 2 to prevent data races and deadlocks in multithreaded programs. The rest of this chapter is organized as follows. Sections 4.1 and 4.2 presents our basic type system for preventing data races and deadlocks. Section 4.3 provides a formal description of the type system. Sections 4.6 and 4.7 present extensions to the basic type system.

SafeJava is primarily a static type system. However, languages like Java allow downcasts that are checked at runtime. Section 4.4 describes an efficient technique for supporting safe runtime downcasts in SafeJava.

Although SafeJava is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information. Section 4.5 describes type inference techniques that significantly reduce programming overhead.

To gain preliminary experience, we implemented several Java programs in SafeJava. Section 4.8 describes our experience in using SafeJava.

Section 4.9 presents related work and Section 4.10 concludes.

SJ1. Every object has an owner.

SJ2. The owner can either be another object or world or thisThread.

SJ3. The ownership relation forms a tree rooted at world.

SJ4. The owner of an object does not change over time.
(Except if there is a unique pointer to that object.)

SJ5. If object z owns y but $z \not\succeq x$, then x cannot access y.
(Except if x is an inner class object of z.)

Figure 4-1: Ownership Properties

SJ8. Objects directly or transitively owned by a thisThread owner are local to the corresponding thread. All other objects are potentially shared between threads.

SJ9. By default, every object is protected by the lock on the root owner of that object. r is the root owner of an object o iff $r \succeq o$ and world directly owns r.
(An object can also be protected by an arbitrary lock. See Section 4.6.3.)

SJ10. To access to an object, a thread must hold the lock that protects that object.
Every thread implicitly holds the lock on the corresponding thisThread owner. A thread can therefore access any object directly or transitively owned by its corresponding thisThread owner without any synchronization.
(Immutable objects and objects with unique pointers can also be accessed without synchronization. See Sections 4.6.4 and 4.6.5.)

Figure 4-2: Properties of Thread-Local and Shared Objects

4.1 Preventing Data Races

This section presents the basic SafeJava type system for preventing data races that supports objects protected by mutual exclusion locks and thread-local objects. Section 4.6 extends the basic type system to make it more expressive.

The key to SafeJava type system is the concept of object ownership. Every object has an owner. Recall the ownership properties from Figure 2-4. To prevent data races, SafeJava statically guarantees the properties in Figures 4-1 and 4-2. Properties SJ1 to SJ5 in Figure 4-1 are similar to those in Figure 2-4, except that an object can also be owned by a special per-thread owner called thisThread. Objects directly or transitively owned by thisThread are local to the corresponding thread and cannot be accessed by any other thread. All other objects are potentially shared between multiple threads.

Figure 4-3 presents an example ownership relation. We draw an arrow from object x to object y if object x owns object y. In the figure, the thisThread owner of Thread 1 is the root owner of objects o_1, o_2, and o_3; the thisThread owner of Thread 2 is the root owner of object o_4; object o_5 is the root owner of objects o_5, o_6, o_7, and o_8; and object o_9 is the root owner of objects o_9 and o_{10}. (Recall from Figure 4-2 that r is the root owner of an object o iff $r \succeq o$ and world directly owns r.)

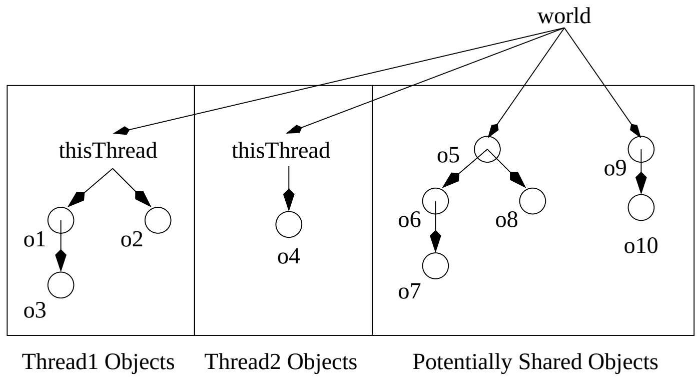

Figure 4-3: An Ownership Relation

Supporting Multithreading

To simplify the presentation of key ideas, we describe our type system as an extension to the core SafeJava type system we presented in Section 2.2 in Chapter 2. A SafeJava program is a sequence of class definitions followed by an initial expression. A predefined class Object is the root of the class hierarchy. Figure 4-4 presents grammar extensions to support multithreading.

Each object in SafeJava has an associated lock that has two states—locked and unlocked—and is initially unlocked. The expression fork($x*$) $\{e\}$ spawns a new thread with arguments ($x*$) to evaluate e. The evaluation is performed only for its effect; the result of e is never used. Note that the Java mechanism of starting threads using code of the form {Thread t=...; t.start();} can be expressed equivalently in SafeJava as {fork(t) {t.start();}}. The expression synchronized (x) in $\{e\}$ works as in Java. x should evaluate to an object. The evaluating thread holds the lock on object x while evaluating e. The value of the synchronized expression is the result of e. While one thread holds a lock, any other thread that attempts to acquire the same lock blocks until the lock is released. A newly forked thread does not inherit locks held by its parent thread.

Each variable and field declaration in SafeJava includes an initialization expression and an optional final modifier. If the modifier is present, then the variable or field cannot be updated after initialization. Other SafeJava constructs are similar to the corresponding constructs in Java.

4.1.1 Owner Polymorphism

Figure 4-5 presents extensions to the grammar shown in Figures 2-5 and 2-6 to prevent data races. (Appendix 4.A at the end of this Chapter presents the complete grammar.) Figure 4-7 shows an example TStack program.[1] For simplicity, all examples in this thesis use a language that is syntactically close to Java. A TStack is a stack of T objects. It is implemented using a linked list.

Every class definition in SafeJava is parameterized with one or more owners. The first owner parameter is special: it identifies the owner of the corresponding object. The other

[1]The example shows type annotations written explicitly. However, many of them can be automatically inferred. We discuss type inference in Section 4.5.

$$
\begin{array}{rcl}
\textit{field} & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ \textit{fd} = e \\
\textit{arg} & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ x \\
e & ::= & ...\ |\ \mathsf{synchronized}\ (x)\ \mathsf{in}\ \{e\}\ |\ \mathsf{fork}\ (x^*)\ \{e\}
\end{array}
$$

Figure 4-4: Grammar Extensions to Support Multithreading

owner parameters are used to propagate ownership information. Parameterization allows programmers to implement a generic class whose objects have different owners. As we described in Section 2.2, an owner can be instantiated with this, with world, or with another owner parameter; in addition, an owner can also be instantiated with thisThread.

In Figure 4-7, the TStack class is parameterized by stackOwner and TOwner. stackOwner owns the TStack object. (Recall that the first owner parameter always owns the corresponding object.) TOwner owns the T objects contained in the TStack. The code specifies that the TStack object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the TStack object. In case of s1, the owner thisThread is used for both the parameters to instantiate the TStack class. It means that TStack s1 as well as all the T objects contained in the TStack are local to the main thread. In case of s2, the TStack is local to the main thread but the T objects contained in the TStack are potentially shared between multiple threads. In case of s3, both the TStack and the T objects contained in the TStack are potentially shared between multiple threads. The objects s2 and s3 are therefore protected by their own locks. (Note that every Java object has an associated lock.) The ownership relation for the TStack objects s1, s2, and s3 is depicted in Figure 4-8 (assuming the stacks contains two elements each). This example illustrates how different TStacks with different protection mechanisms can be created from the same TStack implementation.

Constraints on Owners

As before, for every type $cn\langle o_1, ..., o_n\rangle$ with multiple owners, SafeJava statically enforces the constraint that $(o_i \succeq o_1)$ for all $i \in \{1..n\}$. For a method $m\langle o_{n+1}, ..., o_k\rangle(...)\{...\}$ of an object of type $cn\langle o_1, ..., o_n\rangle$, the constraint is that $(o_i \succeq o_i)$ for all $i \in \{1..k\}$. In addition to enforcing object encapsulation, these constraints prevents thread-local objects from being accessible from objects local to other threads and thread-shared objects. Thus, the type of TStack s4 in Figure 4-7 is illegal because (thisThread $\not\succeq$ world).

The rule for declaring a subtype is as before, that the first owner parameter of the supertype must be the same as that of the subtype and the supertype must satisfy owner constraints.

4.1.2 Requires Clauses

Methods in SafeJava can contain requires clauses to specify the assumptions that hold at method boundaries. Methods specify the objects they access that they assume are protected by externally acquired locks. Callers are required to hold the locks on the root owners of the objects specified in the requires clause before they invoke a method. Consider a method that contains a requires$(o_1, ..., o_n)$ clause. The method can then safely read or write any object x (or call methods that read or write x) where $(o_i \succeq x)$ for some $i \in \{1..n\}$. Callers are required to hold the locks on the root owners of $o_1, ..., o_n$ before they invoke the method.

In the example, the push and pop methods in the TStack class assume that the callers hold the lock on the root owner of the this TStack object. Without the requires clause, the push

$$\begin{array}{rcl} \textit{owner} & ::= & \textit{formal} \mid \textsf{this} \mid \textsf{world} \mid \textsf{thisThread} \\ \textit{meth} & ::= & t\ \textit{mn}\langle \textit{formal}^* \rangle (\textit{arg}^*)\ \textsf{requires}\ (x^*)\ \textsf{where}\ \textit{constr}^*\ \{e\} \end{array}$$

Figure 4-5: Grammar Extensions to Prevent Data Races

```
class Account<accountOwner> {
  int balance = 0;

  int balance()        requires (this) { return balance; }
  void deposit(int x)  requires (this) { balance += x; }
  void withdraw(int x) requires (this) { balance -= x; }
}

Account<thisThread> a1 = new Account<thisThread>;
a1.deposit(10);

Account<world> a2 = new Account<world>;
fork (a2) { synchronized (a2) in { a2.deposit(10); } }
fork (a2) { synchronized (a2) in { a2.deposit(10); } }
```

Figure 4-6: Account

and pop methods would not have been well-typed. All the threads that call the pop method on s2 first acquire the lock on s2. For s1, however, the main thread implicitly holds the lock on the thisThread owner that owns s1. Hence it does not explicitly acquire any locks before calling the pop method on s1.

The requires clauses are similar to the reads and writes clauses we presented in Section 3.1—the requires clauses specify the objects the method reads or writes; and they use the name of an object o to denote all the objects (reflexively and transitively) owned by o. However, the requires clauses are different in that they only have to specify the objects the method reads or writes that are protected by externally held locks. Therefore, if a method acquires the lock that protects an object before reading or writing the object, then the method does not have to specify that object in its requires clause. Similarly, if a method reads or writes a thread-local object, the method does not have to specify that object in its requires clause.

Figure 4-6 presents another example that shows an Account class. The program creates two Account objects a1 and a2. Account object a1 is declared to be thread-local. Account object a2 is declared to be shared between multiple threads and so it is protected by its own lock. The methods of the Account class each have an requires clause that specifies that the methods access the this Account object without synchronization. To prevent data races, the callers of an Account method must hold the lock that protects the corresponding Account object before the callers can invoke the Account methods. Without the requires clauses, the Account methods would not have been well-typed. In the example, all the threads that call the deposit method on a2 first acquire the lock on a2. For a1, however, the main thread implicitly holds the lock on the thisThread owner that owns a1. Hence, it does not explicitly acquire any locks before calling the deposit method on a1.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() {
    if (head == null) return null;
    T<TOwner> value = head.value();  head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  TNode<nodeOwner, TOwner> next;  T<TOwner> value;

  TNode(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next  = n;
  }
  T<TOwner> value()               { return value; }
  TNode<nodeOwner, TOwner> next() { return next;  }
}

class T<TOwner> { }


TStack<thisThread, thisThread> s1;
TStack<thisThread, world>      s2;
TStack<world,      world>      s3;
/* TStack<world,      thisThread> s4; illegal! */


s1.pop();

fork (s2) { synchronized (s2) in { s2.pop(); } }
fork (s2) { synchronized (s2) in { s2.pop(); } }
```

Figure 4-7: Stack of T Objects

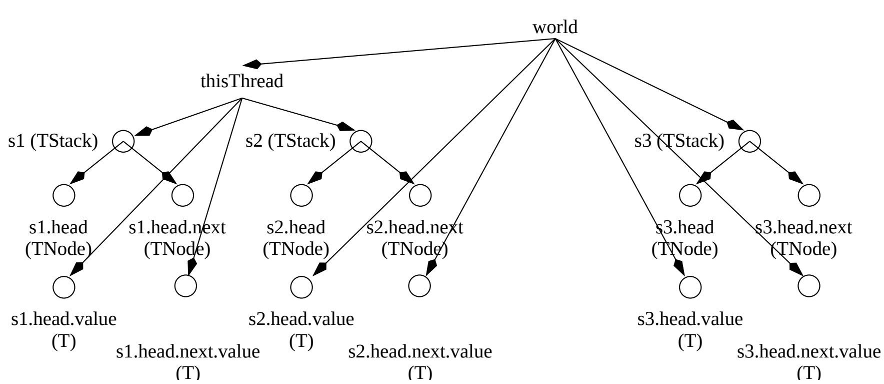

Figure 4-8: Ownership Relation for TStacks s1, s2, s3

$$
\begin{array}{rcl}
\textit{defn} & ::= & \mathsf{class}\ cn\langle formal+\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ constr^*\ \{level^*\ field^*\ meth^*\} \\
\textit{owner} & ::= & formal \mid \mathsf{this} \mid \mathsf{thisThread} \mid \mathsf{world}{:}cn.l \\
\textit{level} & ::= & \mathsf{LockLevel}\ l = \mathsf{new} \mid \mathsf{LockLevel}\ l < cn.l^* > cn.l^* \\
\textit{meth} & ::= & t\ mn\langle formal^*\rangle(arg^*)\ \mathsf{requires}\ (x^*)\ \mathsf{locks}\ (cn.l^*\ [x]_{\mathrm{opt}})\ \mathsf{where}\ constr^*\ \{e\} \\
 & & \\
l & \in & \text{lock level names}
\end{array}
$$

Figure 4-9: Grammar Extensions to Prevent Deadlocks

SJ11. The lock levels form a partial order.

SJ12. Every lock belongs to some lock level. The lock level of a lock does not change over time.
(Except if there is a unique pointer to that lock.)

SJ13. To acquire a new lock of lock level l, the levels of all the locks that the thread currently holds must be greater than l.
(Except if the locks in the lock level l are further ordered. See Section 4.7.)

SJ14. A thread may also acquire a lock that it already holds. The lock acquire operation is redundant in that case.

Figure 4-10: Lock Level Properties

4.2 Preventing Deadlocks

This section presents the basic SafeJava type system for preventing both data races and deadlocks. To prevent deadlocks, programmers specify a partial order among all the locks. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. This section only describes our basic type system that allows programmers to partition the locks into a fixed number of equivalence classes and specify a partial order among the equivalence classes. Our system also allows programmers to use recursive tree-based data structures to describe the partial order—we describe extensions to our basic type system in Section 4.7.

4.2.1 Static Lock Levels

Figure 4-9 presents grammar extensions to statically prevent deadlocks. (Appendix 4.A at the end of this Chapter presents the complete grammar.) SafeJava allows programmers to define lock levels in class definitions. A lock level is like a static field in Java—a lock level is a per-class entity rather than a per-object entity. But unlike static fields in Java, SafeJava uses lock levels only for compile-time type checking and does not preserve them at runtime. Programmers can specify a partial order among the lock levels using the $<$ and $>$ syntax in the lock level declarations. Since a program has a fixed number of lock levels, SafeJava can statically verify that the lock levels do indeed form a partial order. Every lock in SafeJava belongs to some lock level. Note that the set of locks in the language we describe so far is exactly the set of objects that are directly owned by world. A lock is, therefore, an object that has world as its first owner. In SafeJava, every world owner is augmented with the lock level that the corresponding lock belongs to. The properties of our lock levels are summarized in Figure 4-10.

```
class CombinedAccount<immutable> {
  LockLevel savingsLevel = new;
  LockLevel checkingLevel < savingsLevel;

  final Account<world:savingsLevel>  savingsAccount  = new Account;
  final Account<world:checkingLevel> checkingAccount = new Account;
  ...
  void transfer(int x) locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        savingsAccount.withdraw(x); checkingAccount.deposit(x);
  }}}
  int creditCheck() locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance() + checkingAccount.balance();
  }}}
}
```

Figure 4-11: Combined Account

Figure 4-11 presents a CombinedAccount class that uses the Account class shown in Figure 4-6.[2] The program declares the CombinedAccount class to be immutable. A CombinedAccount may not be modified after initialization. The CombinedAccount class contains two Account fields—savingsAccount and checkingAccount. These two Account objects have world as their first owner—these objects are therefore locks. To prevent deadlocks, the CombinedAccount class defines two lock levels—savingsLevel and checkingLevel; and declares that checkingLevel is less than savingsLevel. It further declares that savingsAccount belongs to savingsLevel and checkingAccount belongs to checkingLevel. In the example, both the methods of CombinedAccount acquire locks in the descending order of lock levels by acquiring the lock on savingsAccount before acquiring the lock on checkingAccount.

4.2.2 Locks Clauses

Methods in SafeJava can have locks clauses in addition to requires clauses to specify assumptions at method boundaries. A locks clause contains a set of lock levels. These lock levels are the levels of locks that the corresponding method may acquire. To ensure that a program is free of deadlocks, a thread that calls the method can only hold locks that are of a higher level than the levels specified in the locks clause. In the example in Figure 4-11, both the methods of CombinedAccount contain a locks(savingsLevel) clause. A thread that invokes either of these methods can only hold locks whose levels are greater than savingsLevel.

A locks clause can also contain a lock in addition to lock levels. If a locks clause contains an object l, then a thread that invokes the corresponding method may already hold the lock on object l. Re-acquiring the lock within the method would be redundant in that case. This is useful to support the case where a synchronized method of a class calls another synchronized method of the same class. Figure 4-12 shows part of a self-synchronized Vector implemented in SafeJava. (A self-synchronized class is a class that has world as its first owner instead of a

[2]As we mentioned before, all the examples in this thesis use an extended language even though we present our formalism in the context of a core language.

```
class Vector<world:Vector.l, elementOwner> {
  LockLevel l = new;
  ...
  int elementCount = 0;
  int size()        locks (this) { synchronized (this) { return elementCount;  } }
  boolean isEmpty() locks (this) { synchronized (this) { return (size() == 0); } }
}
```

Figure 4-12: Self-Synchronized Vector

formal owner parameter. Methods of a self-synchronized class can assume that world owns the this object—they can therefore synchronize on this and access the this object without requiring external locks using the requires clause. Section 4.6.1 has more details on self-synchronized classes.) In the example, the isEmpty method acquires the lock on this and invokes size which also acquires the lock on this. The second lock acquire is redundant so it does not violate our condition that threads must acquire locks in the descending order.

4.3 Formal Description

The previous sections presented our grammar. This section describes some of the important rules for type checking. The full set of rules and the complete grammar can be found in Appendix 4.A at the end of this Chapter.

The core of our type system is a set of rules for reasoning about the typing judgment: P; E; ls; $l_{\min} \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. ls describes the set of locks held before e is evaluated. $l_{\min}$ is the minimum level among the levels of all the locks held before e is evaluated. t is the type of e. The judgment $P; E \vdash e : t$ states that e is of type t, while the judgment P; E; ls; $l_{\min} \vdash e : t$ states that e is of type t provided ls contains the necessary locks to safely evaluate e and $l_{\min}$ is greater that the levels of all the locks that are newly acquired when evaluating e.

The typing environment E contains the declared types of variables, the declared owner parameters, the declared constraints among owners, and the declared locks clause. We define the typing environment as follows:

$$E ::= \emptyset \mid E,\ [\text{final}]_{\text{opt}}\ t\ x \mid E,\ \text{owner}\ f \mid E,\ \textit{constr} \mid E,\ \textit{locksclause}$$

We define a lock set as follows, where L(e) is the lock that protects e and x.lock is the lock field stored in the Java object x.

$$ls ::= \mathsf{thisThread} \mid ls,\ x_{\text{final}}.\mathsf{lock} \mid ls,\ \text{L}(e_{\text{final}})$$

We define a minimum lock level as follows, where $\text{LUB}(cn_1.l_1\ ...\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$:

$$l_{\min} ::= \infty \mid cn.l \mid \text{LUB}(cn_1.l_1\ ...\ cn_k.l_k)$$

Note that L(x) and LUB(...) are not computed—they are expressions used as such for type checking. The lock level ∞ denotes that the thread currently holds no locks.

The rule for $\mathsf{fork}(x_{1..n})\ e$ checks the expression e using a lock set that contains $\mathsf{thisThread}$ and is otherwise empty since a new thread does not inherit locks held by its parent. Moreover, the environment E might have some types that contain $\mathsf{thisThread}$. But the owner $\mathsf{thisThread}$ in the parent thread is not the same as the owner $\mathsf{thisThread}$ in the child thread. So, all the $\mathsf{thisThread}$ owners in the environment have to be changed to something else; we use the special owner $\mathsf{otherThread}$ for that.

[EXP FORK]

$$\frac{P;\, E;\, ls;\, l_{\min} \vdash x_i : t_i \qquad g_i = \text{final } t_i[\text{otherThread}/\text{thisThread}]\ x_i \qquad P;\, g_{1..n};\, \mathsf{thisThread};\, \infty \vdash e : t}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{fork}\ (x_{1..n})\ \{e\} : \mathsf{int}}$$

The rule for acquiring a redundant lock x simply checks that x is in the lock set ls. The rule for acquiring a new lock using $\mathsf{synchronized}\ x\ \mathsf{in}\ e$ checks that x is a lock of some level $cn.l$ that is less than $l_{\min}$. If the enclosing method has a locks clause that contains a lock x', then the rule checks that either x is the same object as x', or the level of x is less than the level of x'. The rule then type checks e in an extended lock set that includes x and with $l_{\min}$ set to $cn.l$. A lock is a final variable that is owned by world. The value of a final variable does not change after it has been initialized.

[EXP SYNC]

$$\frac{\begin{array}{c} P;\, E \vdash_{\text{final}} x : cn'\langle \mathsf{world}{:}cn.l\ ...\rangle \qquad P \vdash cn.l < l_{\min} \\ (E = E_1,\, \mathsf{locks}(...\ x'),\, E_2) \Longrightarrow (P;\, E \vdash cn.l < \text{level}(x')) \vee (x' = x) \\ P;\, E;\, ls,\, x;\, cn.l \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{synchronized}\ x\ \mathsf{in}\ e : t}$$

[EXP SYNC REDUNDANT]

$$\frac{\begin{array}{c} x \in ls \\ P;\, E;\, ls;\, l_{\min} \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{synchronized}\ x\ \mathsf{in}\ e : t}$$

Before we proceed further with the rules, we give a formal definition for Lock(e), which is the lock that protects the object e. In the SafeJava type system we described so far, every object is protected by the lock on the root owner of that object. Recall from Figure 4-2 that r is the root owner of an object o iff $r \succeq o$ and world directly owns r. The root owner of an object could be $\mathsf{thisThread}$ or an object owned by world.

[LOCK THISTHREAD]

$$\frac{P;\, E \vdash e : cn\langle \mathsf{thisThread}\ o{*}\rangle}{P;\, E \vdash \text{Lock}(e) = \mathsf{thisThread}}$$

[LOCK WORLD]

$$\frac{P;\, E \vdash e : cn\langle \mathsf{world}{:}cn'.l'\ o{*}\rangle}{P;\, E \vdash \text{Lock}(e) = e.\mathsf{lock}}$$

[LOCK THIS]

$$\frac{P;\, E \vdash e : cn\langle \mathsf{this}\ o_{2..n}\rangle}{P;\, E \vdash \text{Lock}(e) = \text{Lock}(\mathsf{this})}$$

If the owner of an expression is a formal owner parameter, then we cannot determine the root owner of the expression from within the static scope of the enclosing class. In that case, we use $\text{L}(e)$ to denote the root owner of e.

[LOCK FORMAL]

$$\frac{P;\, E \vdash e : cn\langle o_{1..n}\rangle \qquad E = E_1,\, \text{owner } o_1,\, E_2}{P;\, E \vdash \text{Lock}(e) = \text{L}(e)}$$

The rule for accessing field $e.fd$ checks that e is a well-typed expression of some type $cn\langle o_{1..n}\rangle$. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t. If the field is not final, the thread must hold the lock on the root owner of e. Since t is declared inside the class, it might contain occurrences of this and the formal class parameters. When t is used outside the class, the rule renames this with the expression e, and the formal parameters with their corresponding actual parameters.

[EXP REF]

$$\frac{P;\,E;\,ls;\,l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\,E \vdash \mathrm{Lock}(e) = r \\ (P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \wedge (r \in ls)) \vee (P \vdash (\mathsf{final}\ t\ fd) \in cn\langle f_{1..n}\rangle)}{P;\,E;\,ls;\,l_{\min} \vdash e.fd : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{P;\,E;\,ls;\,l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\,E \vdash \mathrm{Lock}(e) = r \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \wedge (r \in ls) \\ P;\,E;\,ls;\,l_{\min} \vdash e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}{P;\,E;\,ls;\,l_{\min} \vdash e.fd = e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method checks that the arguments are of the right type and that the thread holds the locks on the root owners of all final expressions in the requires clause of the method. The rule ensures that $l_{\min}$ is greater than all the levels specified in the locks clause of the method. If the locks clause contains a lock l, the rule ensures that either the level of l is less than $l_{\min}$, or the level of l is equal to $l_{\min}$ and l is in the lock set (in which case re-acquiring l within the method is redundant). The rule appropriately renames expressions and types used outside their declared context.

[EXP INVOKE]

$$\frac{\begin{array}{c}\mathrm{Renamed}(\alpha) \stackrel{\text{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_m/f_m][e_1/y_1]..[e_k/y_k] \\ P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ {}^{j \in 1..k})\ \mathsf{requires}(x*)\ \mathsf{locks}(cn.l*\ [x']_{\mathrm{opt}})\ \mathsf{where}\ constr*\ ...) \in cn\langle f_{1..n}\rangle \\ P;\,E;\,ls;\,l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\,E \vdash o_i \succeq o_1 \qquad P;\,E \vdash \mathrm{Renamed}(constr) \\ P;\,E;\,ls;\,l_{\min} \vdash e_j : \mathrm{Renamed}(t_j) \qquad P;\,E \vdash \mathrm{Lock}(\mathrm{Renamed}(x_i)) = r_i \qquad r_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \qquad x'' = \mathrm{Renamed}(x') \qquad P;\,E \vdash (\mathrm{level}(x'') < l_{\min}) \vee (\mathrm{level}(x'') = l_{\min}) \wedge (x'' \in ls)\end{array}}{P;\,E;\,ls;\,l_{\min} \vdash e.mn\langle o_{(n+1)..m}\rangle(e_{1..k}) : \mathrm{Renamed}(t)}$$

The rule for type checking a method assumes that the thread holds the locks on the root owners of all the final expressions specified in the requires clause. The rules also assumes that for each lock held by the thread, the level of the lock is greater than all the levels specified in the locks clause. If the locks clause of the method contains a lock l, the rule assumes that for each lock held by the thread, either the level of the lock is greater than the level of l, or the lock is the same object as l. The rule then type checks the method body under these assumptions.

$\boxed{P;\,E \vdash method}$

[METHOD]

$$\frac{\begin{array}{c}E' = E,\ \text{owner}\ f_{1..n},\ constr*,\ arg_{1..n},\ \mathsf{locks}(...\ [x]_{\mathrm{opt}}) \qquad P;\,E' \vdash wf \qquad P;\,E' \vdash_{\mathsf{final}} x : t' \\ \forall_{i \in \{1..r\}}\ (P;\,E' \vdash_{\mathsf{final}} x_i : t_i \quad \wedge \quad P;\,E' \vdash \mathrm{Lock}(x_i) = r_i) \\ P;\,E';\,\mathsf{thisThread},\ r_{1..r};\ \mathrm{LUB}(cn_j.l_j\ {}^{j \in 1..k}) \vdash e : t\end{array}}{P;\,E \vdash t\ mn\langle f_{1..n}\rangle(arg_{1..n})\ \mathsf{requires}(x_{1..r})\ \mathsf{locks}(cn_j.l_j\ {}^{j \in 1..k}\ [x]_{\mathrm{opt}})\ \mathsf{where}\ constr*\ \{e\}}$$

The rule for subtyping is the same as in Section 2.3.

$\boxed{P;\,E \vdash t_1 <: t_2}$

[SUBTYPE C]

$$\frac{P;\,E \vdash cn\langle o_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ cn'\langle f_1\ o*\rangle\ ...}{P;\,E \vdash cn\langle o_{1..n}\rangle <: cn'\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P;\,E \vdash t_1 <: t_2 \\ P;\,E \vdash t_2 <: t_3}{P;\,E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]

$$\frac{P;\,E \vdash t}{P;\,E \vdash t <: t}$$

4.4 Safe Runtime Downcasts

In SafeJava, programmers parameterize classes and methods with owners. This enables the writing of generic code that can be used in many different contexts. SafeJava is a primarily static type system. The type checker uses the ownership type annotations to statically ensure the absence of certain classes of errors, but it is usually unnecessary to preserve the ownership information at runtime. However, languages like Java [77] are not purely statically typed languages. Java allows downcasts that are checked at runtime. To support safe runtime downcasts, we must preserve some ownership information at runtime when our type system is used in the context of a language like Java.

There are primarily three techniques for implementing parametric polymorphism in a language like Java. The *type erasure* approach [29, 32] is based on the idea of deleting type parameters (so $\mathsf{Stack}\langle\mathsf{T}\rangle$ erases to Stack). But this approach does not preserve ownership information at runtime, so it is unsuitable for supporting safe runtime downcasts with ownership types. In the *code duplication* approach [3], polymorphism is supported by creating specialized classes/methods, each supporting a different instantiation of a parametric class/method. But since the parameters in ownership types are usually objects, this approach will lead to an unacceptably large number of classes/methods. In the *type passing* approach [112, 132, 131], information on type parameters is explicitly stored in objects and passed to code requiring them. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically small, so adding even a single field to every object increases the size of most objects by a significant fraction.

This section describes an efficient technique for supporting safe runtime downcasts with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. It does not use any interprocedural analysis, so it preserves the separate compilation model of Java. Moreover, our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs [101].

Figure 4-13 presents grammar extensions to support runtime casts. We present the static type checking rules for casts below. Casting an object to a supertype of its declared type is always safe. Casting to a subtype of the declared type requires runtime checking. (We presented the rule for subtyping at the end of Section 4.3.)

[EXPRESSION UPCAST]

$$\frac{P; E; ls \vdash e : c_2 \quad P; E \vdash c_2 <: c_1}{P; E; ls \vdash (c_1)\ e : c_1}$$

[EXPRESSION DOWNCAST (REQUIRES RUNTIME CHECK)]

$$\frac{P; E; ls \vdash e : c_1 \quad P; E \vdash c_2 <: c_1}{P; E; ls \vdash (c_2)\ e : c_2}$$

To support downcasts, SafeJava stores information on type parameters explicitly in objects and passes the information to code requiring the information. Our technique for supporting downcasts efficiently is based on two key observations about the nature of parameterization in ownership types. The remainder of this section is organized as follows. Sections 4.4.1 and 4.4.2 describe the key observations that enable us to support downcasts efficiently. Section 4.4.3 presents our technique for supporting safe downcasts.

$$e \quad ::= \quad ... \mid (cn\langle o_{1..n}\rangle)\ e$$

Figure 4-13: Grammar Extensions to Support Runtime Casts

4.4.1 Downcasts to Types With Single Owners

A key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 4-14. In Line 18, object o1 of declared type Object⟨thisThread⟩ is downcast to type T⟨thisThread⟩, where the owner of the declared type of o1 matches the owner of the type that o1 is being downcast into. Hence, this downcast is safe iff o1 belongs to class T at runtime. It is unnecessary to check ownership information at runtime for this downcast.

In general, for any subtype declaration where all the formal owner parameters in the subtype are included in the supertype, it is not necessary to check ownership information at runtime when an object is downcast from the supertype to the subtype. If the owners of the supertype match the owners of the subtype, then the downcast will be safe iff the object belongs to the appropriate class at runtime (e.g., Lines 18 and 24 in Figure 4-14). If the owners do not match, the downcast will always fail (e.g., Lines 19 and 25 in Figure 4-14).

The primary benefit of this observation is that whenever an object is downcast into a type with a single owner, it is unnecessary to check ownership information at runtime to ensure that the downcast is safe. Since a vast majority of classes in a system with ownership types have single owners, this implies that it is unnecessary to check ownership information at runtime for most of the downcasts. The only classes that usually have multiple owners are collection classes. The only times when it might be necessary to check ownership information at runtime to ensure that the downcast is safe is when an object is downcast into a type with multiple owners (e.g., Lines 21 and 22 in Figure 4-14).

4.4.2 Anonymous Owners

Another key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 4-7. The TStack class in the figure is parameterized by stackOwner and TOwner. However, the owner parameter stackOwner is not used in the static scope where it is visible. Similarly, the owner parameter TOwner for class T is not used in the body of class T. If an owner parameter is not used, it is unnecessary to name the parameter. Our system allows programmers to use ⟨-⟩ for such anonymous owner parameters. Figure 4-15 shows grammar extensions to support anonymous owner parameters. Figure 4-18 shows the TStack example in Figure 4-7 implemented using anonymous owners.

The primary benefit of having anonymous owners is that if an owner parameter of a class is not named, it is unnecessary to store the owner parameter of the class at runtime, or pass the owner parameter to code that uses the class at runtime. In a system with ownership types, the only classes that usually have named owners are collection classes with multiple owners. Examples include Vector⟨-,elementOwner⟩, Hashtable⟨-,keyOwner,valueOwner⟩, etc. But most classes have single owners that are anonymous. It is unnecessary to store ownership information for those classes, or pass ownership information to code that uses those classes. Thus, our system incurs a runtime space and time overhead only for code that uses classes with named owner parameters like the collection classes. The rest of the code has no overhead in our system.

```
class T<stackOwner> {...}
class TStack <stackOwner, TOwner> {...}
class TStack2<stackOwner, TOwner> extends TStack<stackOwner, TOwner> {...}

Locklevel l = new;

Object<thisThread> o1, o2, o3;
...
T<thisThread> t1;
T<world:l>    t2;
...
TStack<thisThread, thisThread>  s1;
TStack<thisThread, world:l>     s2;
...
TStack2<thisThread, thisThread> q1;
TStack2<thisThread, world:l>    q2;
...
t1 = (T<thisThread>) o1;                    // Safe iff o1 belongs to class T
t2 = (T<world:l>)    o2;                    // Compile time error
...
s1 = (TStack<thisThread, thisThread>)  o3; // Requires checking runtime ownership
s2 = (TStack<thisThread, world:l>)     o3; // Requires checking runtime ownership
...
q1 = (TStack2<thisThread, thisThread>) s1; // Safe iff s1 belongs to class TStack2
q2 = (TStack2<thisThread, world:l>)    s1; // Compile time error
```

Figure 4-14: TStack Client Code With Runtime Downcasts

$$defn \quad ::= \quad \ldots \mid \mathsf{class}\ cn\langle\text{-}\ formal^*\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ constr^*\ \{level^*\ field^*\ meth^*\}$$

Figure 4-15: Grammar Extensions to Support Anonymous Owners

4.4.3 Preserving Ownership Information at Runtime

This section describes how SafeJava preserves ownership information at runtime for classes and methods with named owner parameters. The previous sections presented our grammar. Appendix 4.A at the end of this Chapter also presents the grammar, with extensions in Figures 4-13 and 4-15. This section presents the rules for translating a SafeJava program into an equivalent program in a Java-like language without ownership types. If we did not have to support safe runtime downcasts, the translation process would have been simple. We could have converted a SafeJava program into an equivalent Java-like program by simply removing the owner parameters and the effects clauses. However, to support safe runtime downcasts, we must preserve some ownership information in the translation process.

The core of our translation is a set of rules of the form: $(\mathcal{T}[\![C]\!]\ P\ E) = C'$. The rule translates a code fragment C to a code fragment C'. P, the program being checked, is included here to provide information about class definitions. E is an environment containing the formal owner parameters in scope in C. The translated code uses the $\mathsf{\$Owner}$ class shown in Figure 4-16. The $\mathsf{\$Owner}$ class contains two static methods that return objects that represent the $\mathsf{thisThread}$ owner and the world owner respectively. The translation rules are presented in Figure 4-17.

```
public class $Owner {
  public static Object WORLD(String level) { return level.intern();         }
  public static Object THISTHREAD()       { return Thread.currentThread(); }
}
```

Figure 4-16: The $Owner Class

Implementation

This section illustrates with examples how our implementation preserves ownership information at runtime for classes and methods with named owner parameters. If a SafeJava program is well-typed with respect to the rules for static type checking, our implementation translates the program into an equivalent Java program. (Actually, our implementation translates a SafeJava program into Java bytecodes directly. But for ease of presentation, we will describe an equivalent translation into Java code.) The translation mechanism is illustrated in Figures 4-18, 4-19, 4-20, and 4-21. Figure 4-18 shows a TStack class with anonymous owners. Figure 4-19 shows client code that uses the TStack class. Figures 4-20 and 4-21 show the translation of the TStack code and the client code.

Classes: Classes in the translated code contain extra owner fields, one for each named owner parameter. For example, in Figure 4-20, the translated TStack class has an extra $TOwner field. The translated TNode class has two extra fields: $thisOwner and $TOwner. The translated T class has no extra fields because the T class does not have any named owner parameters.

Constructors: Constructors in the translated code contain extra owner arguments, one for each named owner parameter of the class. The constructors in the translated code initialize the owner fields of the class with the owner arguments of the constructor. For example, in Figure 4-20, the constructor for TStack has an extra $TOwner argument. The constructor initializes the $TOwner field of the TStack object from the $TOwner argument.

Allocation Sites: Allocation sites in the translated code must pass extra owner arguments to constructors, one for each named owner parameter of the corresponding class. If the owner is an expression that evaluates to an object, the client code passes the object to the constructor. For example, in Figure 4-20, the push method in TStack passes the this object as the first argument to the TNode constructor. If the owner is a formal parameter, the client code passes the value of the formal parameter stored in one of its extra owner fields. For example, in Figure 4-20, the push method in TStack passes the value stored in the $TOwner field as the second argument to the TNode constructor. If the owner is thisThread or world:*l*, the client code passes the object returned by $Owner.THISTHREAD() or $Owner.WORLD("*l*") to the constructor. For example, in Figure 4-21, the client code creates TStacks s1 and s2 by passing $Owner.THISTHREAD() and $Owner.WORLD("*l*") to the TStack constructor respectively.

Methods: SafeJava handles parameterized methods similar to parameterized classes. It passes owner parameters of methods explicitly as method arguments in the translated code.

Casts: Casts in the translated code not only check that the Java types match, but also that the owners match. For example, in Figure 4-21, in Line 15, the translated code not only checks that o1 is of Java type TStack, but also checks that the owner of the T elements

$$
\begin{aligned}
(\mathcal{T}\llbracket P\rrbracket) &= (\mathcal{T}\llbracket \mathit{defn}^*\ e\rrbracket)\\
&= (\mathcal{T}\llbracket \mathit{defn}\rrbracket\ P)^*\ (\mathcal{T}\llbracket e\rrbracket\ P\ \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket \mathit{defn}\rrbracket\ P) &= (\mathcal{T}\llbracket \textsf{class } cn\langle f_{1..n}\rangle \textsf{ extends } cn'\langle o_{1..n'}\rangle \textsf{ where } \mathit{constr}^*\ \{\mathit{level}^*\ \mathit{field}^*\ \mathit{meth}^*\}\rrbracket\ P)\\
&= \textsf{class } cn \textsf{ extends } cn'\ \{\textsf{Object } \$f_{1..n}\ (\mathcal{T}\llbracket \mathit{field}\rrbracket\ P\ [f_{1..n}])^*\ (\mathcal{T}\llbracket \mathit{method}\rrbracket\ P\ [f_{1..n}])^*\}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket \mathit{defn}\rrbracket\ P) &= (\mathcal{T}\llbracket \textsf{class } cn\langle\text{-}\ f_{2..n}\rangle \textsf{ extends } cn'\langle o_{1..n'}\rangle \textsf{ where } \mathit{constr}^*\ \{\mathit{level}^*\ \mathit{field}^*\ \mathit{meth}^*\}\rrbracket\ P)\\
&= \textsf{class } cn \textsf{ extends } cn'\ \{\textsf{Object } \$f_{2..n}\ (\mathcal{T}\llbracket \mathit{field}\rrbracket\ P\ [f_{2..n}])^*\ (\mathcal{T}\llbracket \mathit{method}\rrbracket\ P\ [f_{2..n}])^*\}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket \mathit{meth}\rrbracket\ P\ E) &= (\mathcal{T}\llbracket t\ mn\langle f_{1..n}\rangle(\mathit{arg}^*) \textsf{ requires } (x^*) \textsf{ locks } (cn.l^*\ [x]_{\mathrm{opt}}) \textsf{ where } \mathit{constr}^*\ \{e\}\rrbracket\ P\ E)\\
&= (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ mn\ (\textsf{Object } \$f_1\ \ldots\ \textsf{Object } \$f_n\ (\mathcal{T}\llbracket \mathit{arg}\rrbracket\ P\ E)^*)\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket \mathit{field}\rrbracket\ P\ E) &= (\mathcal{T}\llbracket [\textsf{final}]_{\mathrm{opt}}\ t\ \mathit{fd} = e\rrbracket\ P\ E)\\
&= [\textsf{final}]_{\mathrm{opt}}\ (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ \mathit{fd} = (\mathcal{T}\llbracket e\rrbracket\ P\ E)
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket \mathit{arg}\rrbracket\ P\ E) &= (\mathcal{T}\llbracket [\textsf{final}]_{\mathrm{opt}}\ t\ \mathit{fd}\rrbracket\ P\ E)\\
&= [\textsf{final}]_{\mathrm{opt}}\ (\mathcal{T}\llbracket t\rrbracket\ P\ E)\ \mathit{fd}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket t\rrbracket\ P\ E) &= (\mathcal{T}\llbracket cn\langle \mathit{owner}+\rangle\rrbracket\ P\ E)\\
&= cn
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket t\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{int}\rrbracket\ P\ E)\\
&= \textsf{int}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket (cn\langle o_{1..n}\rangle)\ e\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = (cn)\ (\mathcal{T}\llbracket e\rrbracket\ P\ E);\\
&\quad\ \textsf{if } (\$\mathrm{temp}.\$f_2\ != (\mathcal{O}\llbracket o_2\rrbracket\ P\ E))\ \textsf{throw new ClassCastException};\\
&\quad\ \ldots;\\
&\quad\ \textsf{if } (\$\mathrm{temp}.\$f_n\ != (\mathcal{O}\llbracket o_n\rrbracket\ P\ E))\ \textsf{throw new ClassCastException};\\
&\quad\ \$\mathrm{temp}\}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{new } cn\langle o_{1..n}\rangle\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = \textsf{new } cn;\\
&\quad\ \$\mathrm{temp}.\$f_1 = (\mathcal{O}\llbracket o_1\rrbracket\ P\ E);\ \ldots;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}\llbracket o_n\rrbracket\ P\ E);\\
&\quad\ \$\mathrm{temp}\}\\
&\quad \text{where } (\textsf{class } cn\langle f_{1..n}\rangle\ \ldots) \in P
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{new } cn\langle o_{1..n}\rangle\rrbracket\ P\ E)\\
&= \{\$\mathrm{temp} = \textsf{new } cn;\\
&\quad\ \$\mathrm{temp}.\$f_2 = (\mathcal{O}\llbracket o_2\rrbracket\ P\ E);\ \ldots;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}\llbracket o_n\rrbracket\ P\ E);\\
&\quad\ \$\mathrm{temp}\}\\
&\quad \text{where } (\textsf{class } cn\langle\text{-}\ f_{2..n}\rangle\ \ldots) \in P
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e_1.mn\langle o_{1..n}\rangle(e^*)\rrbracket\ P\ E)\\
&= (\mathcal{T}\llbracket e_1\rrbracket\ P\ E).mn((\mathcal{O}\llbracket o_1\rrbracket\ P\ E)\ \ldots\ (\mathcal{O}\llbracket o_n\rrbracket\ P\ E)\ (\mathcal{T}\llbracket e\rrbracket\ P\ E)^*)
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \textsf{thisThread}\rrbracket\ P\ E) &&= \$\mathrm{Owner.THISTHREAD}()\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \textsf{world:}cn.l\rrbracket\ P\ E) &&= \$\mathrm{Owner.WORLD}(\texttt{"}cn.l\texttt{"})\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket \textsf{this}\rrbracket\ P\ E) &&= \textsf{this}\\
(\mathcal{O}\llbracket o\rrbracket\ P\ E) &= (\mathcal{O}\llbracket f\rrbracket\ P\ [\ldots\ f\ \ldots]) &&= \$\mathrm{f}
\end{aligned}
$$

$$
\begin{aligned}
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket x\rrbracket\ P\ E) &&= x\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket x = e\rrbracket\ P\ E) &&= x = (\mathcal{T}\llbracket e\rrbracket\ P\ E)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{let } (\mathit{arg}{=}e_1) \textsf{ in } \{e\}\rrbracket\ P\ E) &&= \textsf{let } (\mathit{arg}{=}(\mathcal{T}\llbracket e_1\rrbracket\ P\ E)) \textsf{ in } \{(\mathcal{T}\llbracket e\rrbracket\ P\ E[\mathit{arg}])\}\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e.\mathit{fd}\rrbracket\ P\ E) &&= (\mathcal{T}\llbracket e\rrbracket\ P\ E).\mathit{fd}\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket e_1.\mathit{fd} = e\rrbracket\ P\ E) &&= (\mathcal{T}\llbracket e_1\rrbracket\ P\ E).\mathit{fd} = (\mathcal{T}\llbracket e\rrbracket\ P\ E)\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{synchronized } (x) \textsf{ in } \{e\}\rrbracket\ P\ E) &&= \textsf{synchronized } ((\mathcal{T}\llbracket x\rrbracket\ P\ E)) \textsf{ in } \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}\\
(\mathcal{T}\llbracket e\rrbracket\ P\ E) &= (\mathcal{T}\llbracket \textsf{fork } (x^*)\ \{e\}\rrbracket\ P\ E) &&= \textsf{fork } (x^*)\ \{(\mathcal{T}\llbracket e\rrbracket\ P\ E)\}
\end{aligned}
$$

Figure 4-17: Translation Function

```
class TStack<-, TOwner> {

  TNode<this, TOwner> head = null;

  TStack() {}
  void push(T<TOwner> value) requires (this) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() requires (this) {
    T<TOwner> value = head.value(); head = head.next(); return value;
  }
}

class TNode<thisOwner, TOwner> {

  T<TOwner> value; TNode<thisOwner, TOwner> next;

  TNode(T<TOwner> v, TNode<thisOwner, TOwner> n) requires (this) {
    this.value = v; this.next  = n;
  }
  T<TOwner> value()               requires (this) { return value; }
  TNode<thisOwner, TOwner> next() requires (this) { return next;  }
}

class T<-> { int x=0; }
```

Figure 4-18: TStack With Anonymous Owners

```
class T<-> {...}
class TStack<-, TOwner> {...}
class TStack2<-, TOwner> extends TStack<-, TOwner> {...}

Locklevel l = new;

Object<thisThread> o1;
Object<thisThread> o2;
...
TStack<thisThread, thisThread>  s1 = new TStack<thisThread, thisThread>;
TStack<thisThread, world:l>     s2 = new TStack<thisThread, world:l>;
...
TStack2<thisThread, thisThread> q1;
TStack2<thisThread, world:l>    q2;
...
s1 = (TStack<thisThread, thisThread>)  o1;
s2 = (TStack<thisThread, world:l>)     o2;
...
q1 = (TStack2<thisThread, thisThread>) s1;
q2 = (TStack2<thisThread, world:l>)    s2;
...
boolean b1 = (o1 instanceof TStack<thisThread, thisThread>);
boolean b2 = (o2 instanceof TStack<thisThread, world:l>);
```

Figure 4-19: Client Code for TStack

```
class TStack {
  Object $TOwner;
  TNode head = null;

  TStack(Object $TOwner) { this.$TOwner = $TOwner; }
  void push(T value) {
    TNode newNode = new TNode(this, $TOwner, value, head);
    head = newNode;
  }
  T pop() {
    T value = head.value(); head = head.next(); return value;
  }
}

class TNode {
  Object $thisOwner, $TOwner;
  T value; TNode next;

  TNode(Object $thisOwner, Object $TOwner, T v, TNode n) {
    this.$thisOwner = $thisOwner; this.$TOwner = $TOwner;
    this.value = v; this.next  = n;
  }
  T value()    { return value; }
  TNode next() { return next; }
}

class T { int x=0; }
```

Figure 4-20: Translation of TStack in Figure 4-18

```
class T {...}
class TStack {...}
class TStack2 extends TStack {...}

Object o1;
Object o2;
...
TStack s1 = new TStack($Owner.THISTHREAD());
TStack s2 = new TStack($Owner.WORLD("l"));
...
TStack2 q1;
TStack2 q2;
...
s1 = (TStack) o1;
if (s1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
s2 = (TStack) o2;
if (s2.$TOwner != $Owner.WORLD("l")) throw new ClassCastException();
...
q1 = (TStack2) s1;
if (q1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
q2 = (TStack2) s2;
if (q2.$TOwner != $Owner.WORLD("l")) throw new ClassCastException();
...
boolean b1 = (o1 instanceof TStack) && (((TStack) o1).$TOwner == $Owner.THISTHREAD());
boolean b2 = (o2 instanceof TStack) && (((TStack) o2).$TOwner == $Owner.WORLD("l"));
```

Figure 4-21: Translation of TStack Client Code in Figure 4-19

```
class A<oa1, oa2> {...};
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...};

class C<oc1> {
  void m(B<this, oc1, thisThread> b) {
    A a1;
    B b1;
    b1 = b;
    a1 = b1;
}}
```

Figure 4-22: Incompletely Typed Method

in the TStack is thisThread. In Line 16, the translated code not only checks that o2 is of Java type TStack, but also checks that the owner of the T elements in the TStack is world:l.

InstanceOf: The instanceof operation in the translated code returns true iff the Java types match and the owners match. For example, in Figure 4-21, in Line 20, instanceof returns true iff o1 is of Java type TStack and the owner of the T elements in the TStack is thisThread. In Line 21, instanceof returns true iff o2 is of Java type TStack and the owner of the T elements in the TStack is self.

Arrays: The technique described here does not support safe runtime downcasts to array types. This is because we cannot add extra owner fields to array objects in the translated code and yet remain JVM-compatible. If a programmer wants to downcast from java.lang.Object to an array type in our system, the programmer can create a wrapper object that contains the array object and perform the downcast on the wrapper object.

4.5 Type Inference

Although SafeJava is explicitly typed in principle, it would be onerous to fully annotate every program with the extra type information that SafeJava requires compared to Java. Instead, SafeJava uses a combination of inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. We emphasize that the SafeJava approach to inference is purely intraprocedural and it does not infer method signatures or types of instance variables. Rather, it uses a default completion of partial type specifications in those cases. This approach permits separate compilation.

4.5.1 Intraprocedural Type Inference

In SafeJava, it is usually unnecessary to explicitly augment the types of method-local variables with their owner parameters. A simple inference algorithm can automatically deduce the owner parameters for otherwise well-typed programs. We illustrate our algorithm with an example. Figure 4-22 shows a class hierarchy and an incompletely-typed method m. The types of local variables a1 and b1 inside m do not contain their owner parameters explicitly. The inference algorithm works by first augmenting such incomplete types with the appropriate number of distinct, unknown owner parameters. For example, since a1 is of type A, the algorithm augments the type of a1 with two owner parameters. Figure 4-23 shows augmented types for the example in Figure 4-22. The goal of the inference algorithm

```
 6       A<x1, x2>     a1;
 7       B<x3, x4, x5> b1;


Statement 8   ==>   x3 = this, x4 = oc1, x5 = thisThread
Statement 9   ==>   x1 = x3,   x2 = x5
```

Figure 4-23: Types Augmented With Unknown Owners and Constraints on Owners

is to find known owner parameters that can be used in place of the unknown parameters such that the program becomes well-typed.

The inference algorithm treats the body of the method as a bag of statements. The algorithm works by collecting constraints on the owner parameters for each assignment or function invocation in the method body. Figure 4-23 shows the constraints imposed by Statements 8 and 9 in the example in Figure 4-22. Note that all the constraints are of the form of equality between two owner parameters. Consequently, the constraints can be solved using the standard Union-Find algorithm in almost linear time [45]. If the solution is inconsistent, that is, if any two known owner parameters are constrained to be equal to one another by the solution, then the inference algorithm returns an error and the program does not type check. Otherwise, if the solution is incomplete, that is, if there is no known parameter that is equal to an unknown parameter, then the algorithm replaces all such unknown parameters with thisThread.

4.5.2 Default Types

In addition to supporting intraprocedural type inference, SafeJava provides well-chosen defaults to reduce the number of annotations needed in many common cases. SafeJava also allows user-defined defaults to cover specific sharing patterns that might occur in user code. The following are some default types SafeJava currently provides.

If a programmer declares a class to be default-single-threaded, SafeJava adds the following default type annotations wherever they are not explicitly specified by the programmer. If the type of any instance variable in the class or any method argument or return value is not explicitly parameterized, SafeJava augments the type with an appropriate number of thisThread owner parameters. If a method in the class does not contain a requires or locks clause, SafeJava adds an empty requires or locks clause to the method. With these default types, single-threaded programs require no extra type annotations.

If a programmer declares a class to be default-self-synchronized, SafeJava adds the following default type annotations wherever they are not explicitly specified by the programmer. If the type of any instance variable is not explicitly parameterized, SafeJava augments the type with an appropriate number of this owner parameters. If the type of any method argument or return value is not explicitly parameterized, SafeJava augments the type with fresh formal owner parameters. If a method in the class does not contain a requires clause, SafeJava adds a requires clause that contains all the method arguments. If a method in the class does not contain a locks clause, SafeJava adds a locks(this) clause. With these default types, many self-synchronized classes require almost no extra type annotations.

$$
\begin{array}{lll}
\textit{defn} & ::= & \ldots \mid \\
 & & \textsf{class}\ cn\langle \textsf{thisThread}\ \textit{formal}^*\rangle\ \textsf{extends}\ c\ \textsf{where}\ \textit{constr}^*\ \{\textit{level}^*\ \textit{field}^*\ \textit{meth}^*\} \mid \\
 & & \textsf{class}\ cn\langle \textsf{world}\rangle\ \textsf{extends}\ c\ \textsf{where}\ \textit{constr}^*\ \{\textit{level}^*\ \textit{field}^*\ \textit{meth}^*\}
\end{array}
$$

Figure 4-24: Grammar Extensions to Support Self-Synchrinized and Thread-Local Classes

```
class SharedAccount<world:SharedAccount.l> {
  LockLevel l = new;
  int balance = 0;
  int  balance()       requires () locks(l) { synchronized (this) { return balance; } }
  void deposit(int x)  requires () locks(l) { synchronized (this) { balance += x;   } }
  void withdraw(int x) requires () locks(l) { synchronized (this) { balance -= x;   } }
}

SharedAccount<world:SharedAccount.l> a = new SharedAccount<world:SharedAccount.l>;
fork (a) { a.deposit(10); }
fork (a) { a.deposit(10); }
```

Figure 4-25: Self-Synchronized Account

4.6 Extensions for Preventing Data Races

This section presents extensions to our basic race-free type system to make it more expressive.

4.6.1 Self-Synchronized Classes

Sometimes, programmers want to specify in a class declaration that instances of the class are always protected by their own locks. Consider, for example, a SharedAccount class in Figure 4-25 where the deposit method is synchronized, so that the callers of the deposit method do not have to acquire any locks. If a SharedAccount object was owned by some other object, then it would have been necessary to hold the lock on the root owner of the SharedAccount object to access the SharedAccount object. This is because some other thread might acquire the lock on the root owner and access the balance field of the SharedAccount object directly. Thus, the deposit method with the empty requires clause type checks only if the SharedAccount class is declared to be always owned by world. To enable this, SafeJava allows the first parameter in a class declaration to be world. This feature lets us implement self-synchronized classes in SafeJava. Figures 4-12 and 4-25 show examples.

Note that it is highly unusual to have a type system where a constant value is used instead of a formal parameter. But it is necessary in our case because the first parameter in our system is special, in that it owns the this object.

4.6.2 Thread-Local Classes

SafeJava also allows the first parameter in a class declaration to be thisThread. Such a class can only be instantiated with thisThread as the first owner. All instances of such classes would be thread-local. Figure 4-24 presents grammar extensions to support self-synchronized classes and thread-local classes.

$$owner \quad ::= \quad formal \mid \mathsf{this} \mid \mathsf{world} \mid \mathsf{thisThread} \mid e_{\mathrm{final}}.\mathsf{lock}$$

Figure 4-26: Grammar Extensions to Support Objects Protected by Arbitrary Locks

```
class Data<f> { int x = 0; int get reads(this) {return x;} }
Locklevel l = new;
final DataLock<world:l> dl = new DataLock;

Data<dl.lock> data;
fork(data,dl) { synchronized (dl) { data.get(); } }
fork(data,dl) { synchronized (dl) { data.get(); } }
```

Figure 4-27: Object Protected by an Arbitrary Lock

4.6.3 Objects Protected By Arbitrary Locks

In type system we described so far, every (unencapsulated) object owned by world is protected by its own lock (Property SJ9 in Figure 4-2). We chose this as a default because it matches the common programming practice in Java. However, programmers sometimes want to protect an unencapsulated object by an arbitrary lock. To support this, SafeJava allows the owner of an object o to be instantiated by e.lock. It means that the object o is owned by world but is protected by the lock on object e. Figure 4-26 presents grammar extensions to support this. e must be a final expression owed by world. A final expression is either a final variable, or a field $e'.fd$ where e' is a final expression and fd is a final field. Figure 4-27 shows an unencapsulated object data is protected by an arbitrary lock dl.

4.6.4 Objects With Unique Pointers

We did not discuss unique pointers in this chapter for clarity of presentation of the key ideas. But as we presented in Section 3.3 of Chapter 2, SafeJava supports objects with unique pointers. If a thread has a unique (external) pointer to an object, SafeJava allows the thread to read or write the object without any synchronization.

Issues Related to the Java Memory Model

Note that synchronization operations in Java are used not just for mutual exclusion, but also to enforce visibility in multiprocessor machines [117]. Therefore, if Thread 1 creates or updates an object x and passes the unique reference to x to Thread 2 without using synchronization, then updates made by Thread 1 are not guaranteed to be visible to Thread 2. But this is not a problem in SafeJava because the only way Thread 1 can pass the unique reference to x to Thread 2 is by writing the unique reference into a shared data structure that can be subsequently read by Thread 2. But since the shared data structure can only be accessed with synchronization, the updates made by Thread 1 will be visible to Thread 2.

4.6.5 Immutable Objects

As described in Sections 3.4 and 3.1, SafeJava supports immutable objects and reads/writes clauses. SafeJava allows a thread to read an immutable object without synchronization.

$$
\begin{array}{rcl}
\textit{formal} & ::= & f \mid \mathsf{world}{:}v \\
\textit{locklevel} & ::= & cn.l \mid v \\
\textit{constr} & ::= & ... \mid (\textit{locklevel} > \textit{locklevel})^* \\
\textit{locksclause} & ::= & \mathsf{locks}\ (\textit{locklevel}^*\ [\textit{lock}]_{\mathrm{opt}}) \\
\\
v & \in & \text{formal lock level names}
\end{array}
$$

Figure 4-28: Grammar Extensions to Support Lock Level Polymorphism

```
class Stack<world:v, elementOwner> where (v > Vector.l) {
  Vector<world:Vector.l, elementOwner> vec = new Vector;
  ...
  int size() locks(this) {
    synchronized (this) { return vec.size(); }
}}
```

Figure 4-29: Self-Synchronized Stack Implemented Using Vector

4.7 Extensions for Preventing Deadlocks

This section presents extensions our basic deadlock-free type system to make it more expressive.

4.7.1 Lock Level Polymorphism

This section describes how our type system supports polymorphism in lock levels. In the type system described in Section 4.2, the level of each lock is known at compile-time. But programmers sometimes want to write code where the exact levels of some locks are not known statically—only some ordering constraints among the unknown lock levels are known statically. Lock level polymorphism enables this kind of programming. Figure 4-28 shows the grammar extensions to support lock level polymorphism. Programmers can parameterize classes with formal lock level parameters in addition to formal owner parameters. Programmers can specify ordering constraints among the lock level parameters using where clauses. Figure 4-29 shows part of a self-synchronized Stack implemented using the self-synchronized Vector in Figure 4-12. The lock level of the this Stack object is a formal parameter v. The where clause constrains v to be greater than Vector.l. It is therefore legal for the synchronized Stack.size method to call the synchronized Vector.size method. The type checker verifies that the program acquires the locks in the descending order.

4.7.2 Condition Variables

This section describes how SafeJava prevents deadlocks in the presence of condition variables. Java provides condition variables in the form of wait and notify methods on Object. Since a thread can wait on a condition variable as well as on a lock, it is possible to have a deadlock that involves condition variables as well as locks. There is no simple rule like the ordering rule for locks that can avoid this kind of deadlock. The lock ordering rule depends on the fact that a thread must be holding a lock to keep another thread waiting for that lock. In the case of conditions, the thread that will notify cannot be distinguished in such a simple way.

$$\textit{locksclause} \quad ::= \quad \mathsf{locks}\ ([\infty]_{\text{opt}}\ \textit{locklevel*}\ [\textit{lock}]_{\text{opt}})$$

$$e \quad ::= \quad \ldots \mid e.\mathsf{wait} \mid e.\mathsf{notify}$$

Figure 4-30: Grammar Extensions to Support Condition Variables

$$\textit{field} \quad ::= \quad [\mathsf{final}]_{\text{opt}}\ [\mathsf{tree}]_{\text{opt}}\ t\ \textit{fd} = e$$

Figure 4-31: Grammar Extensions to Support Tree Ordering

Figure 4-30 shows the grammar extensions to support condition variables. The expression e.wait and e.notify are similar to the wait and notifyAll methods in Java. e must be a final expression that evaluates to an object, and the current thread must hold the lock on e. On executing wait, the current thread releases the lock on e and suspends itself. The thread resumes execution when some other thread invokes notify on the same object. The thread re-acquires the lock on e before resuming execution after wait.

To prevent deadlocks in the presence of condition variables, SafeJava enforces the following constraint. A thread can invoke e.wait only if the thread holds no locks other than the lock on e. Since a thread releases the lock on e on executing e.wait, the above constraint implies that any thread that is waiting on a condition variable holds no locks. This in turn implies that there cannot be a deadlock that involves a condition variable. To statically verify that a program respects the above constraint, SafeJava requires that any method m that contains a call to e.wait must have a locks (∞) clause or a locks $(\infty\ e)$ clause. The former locks clause indicates that a thread holds no locks when it invokes m, while the later locks clause indicates that a thread can only hold the lock on e when it invokes m. Within the method, SafeJava ensures when type checking e.wait that the lock set only contains the lock on e. The rules for type checking are shown below.

[EXP WAIT]

$$\frac{E = E_1,\ \mathsf{locks}(\infty\ [e]_{\text{opt}}),\ E_2 \qquad P; E \vdash_{\text{final}} e \qquad ls = \{e\}}{P; E; ls; l_{\min} \vdash e.\mathsf{wait} : \mathsf{int}}$$

[EXP NOTIFY]

$$\frac{P; E \vdash_{\text{final}} e \qquad e \in ls}{P; E; ls; l_{\min} \vdash e.\mathsf{notify} : \mathsf{int}}$$

4.7.3 Tree-Based Partial Orders

This section describes how SafeJava supports tree-based partial orders. Figure 4-31 shows the grammar extensions to support tree-based partial orders. Programmers can declare fields in objects to be tree fields. If object x has a tree field fd that contains a pointer to object y, we say that there is a tree edge fd from x to y. x is the parent of y and y is a child of x. SafeJava ensures that the graph induced by the set of all tree edges in the heap is indeed a forest of trees. Any data structure that has a tree backbone can be used to describe the partial order in our system. This includes doubly linked lists, trees with parent pointers, threaded trees, and balanced search trees.

Locks that belong to the same lock level are further ordered according to the tree order. Suppose x and y are two locks (that is, they are objects that are owned by world) that belong to the same lock level. Suppose a thread t holds the lock on x and reads a tree field fd of x to get a pointer to y. So y is a child of x. SafeJava then allows thread t to also acquire the lock on y while holding the lock on x. Note that as long as t holds the lock

```
class BalancedTree {
  LockLevel l = new;
  Node<world:l> root = new Node;
}
class Node<world:k> {
  tree Node<world:k> left, right;

  //   this              this
  //   /  \              /  \
  // ...   x          ...   v
  //      / \    -->       / \
  //     v   y            u   x
  //    / \                  / \
  //   u   w                w   y

  synchronized void rotateRight() locks(this) {
    final Node x = this.right; if (x == null) return;
    synchronized (x) {
      final Node v = x.left; if (v == null) return;
      synchronized (v) {
        final Node w = v.right;
        v.right = null;
        x.left = w;
        this.right = v;
        v.right = x;
  }}} ...
}
```

Figure 4-32: Balanced Tree

on x, no other thread can modify x, so no other thread can make y not a child of x. The type checking rule is shown below, assuming that for every pair of final variables x and y, environment E contains information about whether the objects x and y are related by tree edges.

[EXP SYNC CHILD]

$$\frac{\begin{array}{c}\forall_{y \in ls}\ P; E \vdash (\text{level}(y) > l_{\min}) \vee (y \text{ is an ancestor of } x) \\ x' \in ls \quad P; E \vdash x \text{ is a child of } x' \quad P; E \vdash \text{level}(x) = \text{level}(x') = l_{\min} \\ P; E; ls, x; l_{\min} \vdash e : t\end{array}}{P; E; ls; l_{\min} \vdash \textsf{synchronized } x \textsf{ in } e : t}$$

Figure 4-32 presents an example with a tree-based partial order. The Node class is self-synchronized, that is, the this Node object is owned by world. The lock level of the this Node object is the formal parameter k. A Node has two tree fields left and right. The Nodes left and right own themselves and also belong to lock level k. Nodes left and right are therefore ordered less than the this Node object in the partial order. In the example, the rotateRight method acquires the locks on Nodes this, x, and v in the tree order.

SafeJava allows a limited set of mutations on trees at runtime. The type checker uses a simple intraprocedural intra-loop flow-sensitive analysis to check that the mutations do not introduce cycles in the trees. We illustrate our flow-sensitive analysis using the example in Figure 4-32. The type checker keeps the following additional information in the environment E for every pair of final variables x and y: 1) If the objects x and y are related

Stmt #	Information in Environment After Checking That Statement in Figure 4-32		
21	x=this.right, v=x.left, w=v.right		
22	x=this.right, v=x.left	w is Root	(this, x, v) not in Tree(w)
23	x=this.right, w=x.left	v is Root	(this, x, w) not in Tree(v)
24	v=this.right, w=x.left	x is Root	(this, v) not in Tree(x)
25	v=this.right, w=x.left, x=v.right		

Figure 4-33: Illustration of Flow-Sensitive Analysis

$$\textit{field} \quad ::= \quad [\textsf{final}]_{\text{opt}}\ [\textsf{tree}]_{\text{opt}}\ t\ fd = e \mid \textsf{final dag}\ t\ fd = e$$

Figure 4-34: Grammar Extensions to Support DAG Ordering

by a tree edge, 2) If x is the root of a tree, and 3) If x is a root and y is not in the tree rooted at x. Figure 4-33 contains the information stored in the environment after the type checking of various statements in the rotateRight method in Figure 4-32. Since the analysis is flow-sensitive, the environment changes after checking each statement.

The rules for mutating a tree are as follows. Deleting a tree edge (for example, setting a tree field to null or over-writing a tree field) requires no extra checking. A tree edge from x to x' may be added only if x' is the root of a tree and x is not in the tree rooted at x'. The rule is shown below. Note that if x' is a unique pointer to an object (for example, x' is newly created), then x' is trivially a root. Similarly, if a local variable x contains a unique pointer, then x cannot be in the tree rooted at x'.

[EXP TREE ASSIGN]

$$\frac{\begin{array}{c} P \vdash (\textsf{tree}\ t\ fd) \in cn\langle f_{1..n}\rangle \qquad P;\, E;\, ls;\, l_{\min} \vdash x : cn\langle o_{1..n}\rangle \qquad P;\, E \vdash \text{RootOwner}(x) = r \qquad r \in ls \\ P;\, E;\, ls;\, l_{\min} \vdash x' : t[x/\textsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\, E \vdash x' \text{ is Root} \quad P;\, E \vdash x \text{ not in Tree}(x') \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash x.fd = x' : t[x/\textsf{this}][o_1/f_1]..[o_n/f_n]}$$

4.7.4 DAG-Based Partial Orders

SafeJava also allows programmers to use directed acyclic graphs (DAGs) to describe the partial order. Figure 4-34 shows the grammar extensions to support DAG-based partial orders. Programmers can declare fields in objects to be dag fields. SafeJava ensures that no object can be both part of a tree and part of a DAG. Locks that belong to the same lock level are further ordered according to the DAG-order. DAGs used for partial orders are monotonic. DAG fields cannot be modified once initialized. Only newly created nodes may be added to a DAG by initializing the newly created nodes to contain DAG edges to existing DAG nodes. The rule is shown below.

[EXP DAG ASSIGN]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash x : cn\langle o_{1..n}\rangle \qquad P \vdash (\textsf{dag}\ t\ fd) \in cn\langle f_{1..n}\rangle \qquad P;\, E \vdash \text{RootOwner}(x) = r \qquad r \in ls \\ P;\, E;\, ls;\, l_{\min} \vdash x' : t[x/\textsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\, E \vdash x \text{ is newly created} \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash x.fd = x' : t[x/\textsf{this}][o_1/f_1]..[o_n/f_n]}$$

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \textsf{class}\ cn\langle \mathit{formal+}\rangle\ \mathit{defn}' \mid \textsf{class}\ cn\langle\text{-}\ \mathit{formal*}\rangle\ \mathit{defn}' \mid \\
 & & \textsf{class}\ cn\langle \textsf{thisThread}\ \mathit{formal*}\rangle\ \mathit{defn}' \mid \textsf{class}\ cn\langle \textsf{world}\rangle\ \mathit{defn}' \\
\mathit{defn}' & ::= & \textsf{extends}\ c\ [\textsf{implements Dynamic}]_{\mathrm{opt}}\ \textsf{where}\ \mathit{constr*}\ \{\mathit{level*}\ \mathit{field*}\ \mathit{meth*}\} \\
e & ::= & ... \mid \textsf{synchronized}\ (x+)\ \textsf{in}\ \{e\}
\end{array}
$$

Figure 4-35: Grammar Extensions to Support Runtime Ordering

```
class Account implements Dynamic {
  int balance = 0;

  int balance()        requires (this) { return balance; }
  void deposit(int x)  requires (this) { balance += x; }
  void withdraw(int x) requires (this) { balance -= x; }
}

void transfer(Account<world:v> a1, Account<world:v> a2, int x) locks(v) {
  synchronized (a1, a2) { a1.withdraw(x);  a2.deposit(x); }
}
```

Figure 4-36: Runtime Ordered Accounts

4.7.5 Runtime Ordering of Locks

In the type system we described so far, the partial order between locks is known statically. However, programmers sometimes want to write code where the order cannot be determined statically. For example, consider a transfer method that receives two self-synchronized Account objects a1 and a2. The transfer method acquires the locks on a1 and a2 and transfers money from a1 to a2. But the ordering between a1 and a2 may not be known statically within the transfer method. To avoid deadlocks in such programs, SafeJava supports imposing an arbitrary linear order at runtime on a group of unordered locks. SafeJava also provides a primitive to acquire such locks in the linear order.

Figure 4-35 shows the grammar extensions to support runtime ordering of locks. Programmers can declare a class to be a subtype of Dynamic. Objects of such classes cannot contain tree or dag edges to other objects. The SafeJava runtime imposes an arbitrary linear order on Dynamic objects by assigning a unique id to each of them. The runtime stores the unique id in every Dynamic object.

Locks of type Dynamic that belong to the same lock level are further ordered based on the linear order. SafeJava provides a primitive to acquire multiple Dynamic locks of the same lock level: synchronized(l_1, ..., l_n). To prevent deadlocks, the runtime sorts the locks $l_1 ... l_n$ based on the linear order and acquires the locks in the sorted order.[3] For example, in Figure 4-36, the locks a1 and a2 are of type Dynamic and belong to the same lock level. The synchronized statement acquires the locks in the linear order and thus avoids causing deadlocks.

[3] Our implementation of this feature runs on regular JVMs. We translate a synchronized statement with multiple locks into code that acquires the locks individually in the linear order. We also translate the code in constructors of Dynamic objects to store the unique ids in the objects.

Program	Lines of Code	Lines Changed
Multithreaded Server Programs		
SMTP	2105	46
POP3 Mail	1364	31
elevator	0523	15
http	0563	26
chat	0308	22
stock quote	0242	12
game	0087	11
phone	0302	10
Collection Classes		
java.util.Vector	0992	35
java.util.ArrayList	0533	18
java.util.Hashtable	1011	53
java.util.HashMap	0852	46
Other Library Classes		
java.io.PrintStream	568	14
java.io.FilterOutputStream	148	05
java.io.OutputStream	134	03
java.io.BufferedWriter	253	09
java.io.OutputStreamWriter	266	11
java.io.Writer	177	06

Figure 4-37: Programming Overhead

4.8 Programming Experience

We have a prototype implementation of our type system. Our implementation is JVM-compatible. It translates well-typed programs into bytecodes that can run on regular JVMs.

One of the challenges in designing an effective type system is to make it expressive enough to support common programming paradigms. To gain preliminary experience, we implemented a number of Java programs in SafeJava including several classes from the Java libraries. We also implemented some multithreaded server programs including an *SMTP* server from apache, a *POP3 Mail* server from Apache, *elevator*, a real time discrete event simulator [133, 38], an *http* server, a *chat* server, a *stock quote* server, a *game* server, and *phone*, a database-backed information sever. These programs exhibit a variety of sharing patterns. We found that SafeJava is expressive enough to support the above-mentioned programs. In each case, once we determined the sharing pattern of the program, adding the extra type annotations was fairly straight-forward.

Figure 4-37 shows the lines of code that needed explicit type annotations for some of the programs we implemented in SafeJava. As described in Section 4.5.1, SafeJava infers the owner parameters of method-local variables. Moreover, the defaults provided by the system (described in Section 4.5.2) are sufficient in most cases. On average, we had to annotate about one in thirty lines of code.

In our experience, we found that threads rarely need to hold multiple locks at the same time. In cases where they do, the threads usually acquire the multiple locks as they cross abstraction boundaries. For example, in *elevator*, threads acquire the lock on a Floor object and then invoke synchronized methods on a Vector object. Even though such programs use an unbounded number of locks, these locks can be classified into a small number of lock levels. These programs are therefore easily expressed in SafeJava.

We also note that in cases where threads do hold multiple locks simultaneously, it is usually because of conservative programming. For example, in the PrintStream class in Sun's implementation, the print(String) method acquires the lock on the PrintStream object and then calls a method that acquires the lock on a BufferedWriter object contained within the PrintStream object. Acquiring the second lock is unnecessary and our implementation avoids this. As another example, in the *elevator* example mentioned above, the Vector object is contained within the Floor object. Acquiring the lock on the Vector object is thus unnecessary. In fact, programmers can use an ArrayList instead of a Vector.

The reason many Java programs are conservative is because there is no mechanism in Java to prevent data races or deadlocks. For example, Java programs that use ArrayLists risk data races because ArrayLists may be accessed without appropriate synchronization in shared contexts. But because SafeJava guarantees data race freedom and deadlock freedom, programmers can employ aggressive locking disciplines without sacrificing safety.

Limitations

Our experience suggests that SafeJava is sufficiently expressive to accommodate the commonly used synchronization patterns. However, we did encounter the following limitations.

Static Variables:
Java has global (static) variables that are accessible to all threads. If a program accesses static variables without synchronization, SafeJava cannot verify that this will not lead to data races. Therefore, in SafeJava, a thread can access a static variable only when it holds the lock on the Java class that contains the static variable.

Multithreaded Scientific Programs:
We looked at some scientific programs like barnes and water from the SPLASH-2 benchmark set [136]. These programs proceed through phases that are separated by barriers. Within each phase, there are unsynchronized accesses to disjoint elements of the same array by different threads. SafeJava does not support these synchronization patterns. To accommodate such programs, a type system would have to provide a way for expressing temporal properties—like the fact that two consecutive phases in the program do not overlap in time.

4.9 Related Work

There has been much research on approaches to detect or prevent data races and deadlocks in multithreaded programs.

Static Tools:
Tools like Warlock [127] and Sema [92] use annotations supplied by programmers to statically detect potential data races and deadlocks in a program. The Extended Static Checker for Java (ESC/Java) [57] is another annotation based system that uses a theorem prover

to statically detect many kinds of errors including data races and deadlocks. [63] assumes bugs to be deviant behavior to statically extract and check correctness conditions that a system must obey without requiring programmer annotations. While these tools are useful in practice, they are not sound, in that they do not certify that a program is race-free or deadlock-free. For example, ESC/Java does not always verify that a partial order of locks declared in a program is indeed a partial order.

Dynamic Tools:
There are many systems that detect data races and deadlocks dynamically. These include systems developed in the scientific parallel programming community [59, 35], tools like Eraser [123], and tools for detecting data races in Java programs [133, 38]. Eraser dynamically monitors all lock acquisitions to test whether a linear order exists among the locks that is respected by every thread. Dynamic tools have the advantage that they can check unannotated programs. However, they are not comprehensive—they may fail to detect certain errors due to insufficient test coverage. Besides, annotated programs are easier to understand and maintain because they explicitly contain the design decisions made by programmers.

Language Mechanisms:
To our knowledge, Concurrent Pascal is the first race-free programming language [31]. Programs in Concurrent Pascal use synchronized monitors to prevent data races. But monitors in Concurrent Pascal are restricted in that threads can share data with monitors only by copying the data. A thread cannot pass a pointer to an object to a monitor.

More recently, researchers have proposed type systems for object-oriented programs that guarantee that any well-typed program is free of data races [68, 66, 67, 11]. The work on Race Free Java [68] is closest to ours. Race Free Java extends the static annotations in Esc/Java into a formal type system. It also supports the use of thread-local objects by providing thread-local classes. Instances of thread-local classes need no synchronization. SafeJava builds on this by letting programmers write generic code to implement a class, and create different objects of the same class that have different protection mechanisms. For example, in SafeJava, programmers can write a generic Queue implementation, then create Queue objects that have different protection mechanisms. These different objects could include thread-local Queue objects, shared Queue objects, Queue objects contained within other enclosing data structures, Queue objects containing thread-local items, Queue objects containing shared items, and Queue objects containing items enclosed within other data structures. In Race Free Java, one needed a different Queue implementation to support each of the above cases. Race Free Java also does not support objects with unique pointers or immutable objects that can be accessed without synchronization.

Guava [11] is another dialect of Java for preventing data races. It allows programmers to access objects without synchronization in many common cases where the absence of synchronization does not lead to data races. Guava splits the class hierarchy into three distinct sub-hierarchies. Instances of Monitor classes are self-synchronized shared objects that correspond to the roots of ownership trees in our system. Instances of Object classes are either thread-local or contained within some Monitor. These instances correspond to objects that are either owned by thisThread or by some other object in our system. Instances of Value classes are somewhat analogous to objects with unique pointers in our system. Again, the primary difference between the Guava approach and our approach is that our system lets programmers to write generic code, then create objects that have different protection mechanisms from the same generic code.

None of the above type systems prevent deadlocks. SafeJava statically prevents both data races and deadlocks in multithreaded programs.

Synchronization Removal:
There has been a lot of work recently on compiler analysis techniques to eliminate unnecessary synchronizations [5, 15, 16, 37, 119, 135]. In SafeJava, the natural way to implement most library classes (like a Hashtable, for example) is to require external synchronization. This has the effect of moving synchronization operations up the call chain. This in turn helps programmers structure their programs such that locks are acquired only when necessary. Syntactic sugar can be provided to make it more convenient to acquire the lock on an object before invoking a method on it. Thus, SafeJava provides an alternate way to reduce the number of unnecessary synchronization operations in a program without risking data races.

Message Passing Systems:
There are several systems that statically check for data races and deadlocks in message passing systems [88, 34]. These systems, however, use a different programming model. For example, programs in these systems do not access shared objects in a heap.

4.10 Conclusions

Multithreaded programming is difficult and error prone. This chapter describes how SafeJava statically prevents data races and deadlocks in multithreaded programs. SafeJava combines object encapsulation and safe multithreading in a unified type system framework.

To prevent data races, programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. The SafeJava type system for preventing data races is significantly more expressive than previous proposed type systems. In particular, it lets programmers write generic code to implement a class, then create different objects from the same class that have different protection mechanisms.

To prevent deadlocks, SafeJava allows programmers to partition the locks into a fixed number of lock levels and specify a partial order among the lock levels. SafeJava also allows programmers to use recursive tree-based data structures to further order locks within a given lock level. SafeJava statically verifies that mutations to trees used for describing the partial order do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime.

SafeJava uses an efficient mechanism for supporting safe runtime downcasts.

SafeJava provides type inference and default types that significantly reduce the burden of writing the extra type annotations. In particular, single-threaded programs require almost no programming overhead.

We implemented several multithreaded Java programs in our system. Our experience indicates that SafeJava is sufficiently expressive to express common sharing patterns and requires little programming overhead.

4.A Rules for Type Checking

This section formally presents our basic type system for preventing data races and deadlocks. It builds on the type system presented in Appendix 2.A at the end of Chapter 2. The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & \textit{defn* e} \\
\textit{defn} & ::= & \mathsf{class}\ cn\langle \textit{formal+}\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ \textit{constr*}\ \{\textit{level* field* meth*}\} \\
c & ::= & cn\langle \textit{owner+}\rangle \mid \mathsf{Object}\langle \textit{owner}\rangle \\
\textit{owner} & ::= & \textit{formal} \mid \mathsf{this} \mid \mathsf{world}{:}cn.l \mid \mathsf{thisThread} \\
\textit{constr} & ::= & (\textit{owner} \succeq \textit{owner}) \mid (\textit{owner} \not\succeq \textit{owner}) \\
\textit{level} & ::= & \mathsf{LockLevel}\ l = \mathsf{new} \mid \mathsf{LockLevel}\ l < cn.l* > cn.l* \\
\textit{meth} & ::= & t\ mn\langle \textit{formal*}\rangle(\textit{arg*})\ \mathsf{requires}\ (x*)\ \mathsf{locks}\ (cn.l*\ [x]_{\text{opt}})\ \mathsf{where}\ \textit{constr*}\ \{e\} \\
\textit{field} & ::= & [\mathsf{final}]_{\text{opt}}\ t\ fd = e \\
\textit{arg} & ::= & [\mathsf{final}]_{\text{opt}}\ t\ x \\
t & ::= & c \mid \mathsf{int} \\
\textit{formal} & ::= & f \\
 & & \\
e & ::= & \mathsf{new}\ c \mid x \mid x = e \mid \mathsf{let}\ (\textit{arg}{=}e)\ \mathsf{in}\ \{e\} \mid e.fd \mid e.fd = e \mid e.mn\langle \textit{owner*}\rangle(e*) \mid \\
 & & \mathsf{synchronized}\ (x)\ \mathsf{in}\ \{e\} \mid \mathsf{fork}\ (x*)\ \{e\} \\
 & & \\
cn & \in & \text{class names} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
x,y & \in & \text{variable names} \\
f & \in & \text{owner names} \\
l & \in & \text{lock level names}
\end{array}
$$

We first define a number of predicates used in the type system. These are based on similar predicates from [70, 68]. We refer the reader to those papers for their precise formulation.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
FieldsOnce(P)	No class contains two fields, declared or inherited, with same name
MethodsOncePerClass(P)	No class contains two methods with same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden. The read and write effects of an overriding method must be superseded by those of the overridden methods
LockLevelsOK(P)	There are no cycles in the lock levels

We define the typing environment as follows:

$E ::= \emptyset \mid E,\ [\text{final}]_{\text{opt}}\ t\ x \mid E,\ \text{owner}\ f \mid E,\ \textit{constr} \mid E,\ \textit{locksclause}$

The typing environment contains the declared types of variables, the declared owner parameters, the declared constraints among owners, and the declared locks clause.

We define a lock set as follows: $ls ::= \mathsf{thisThread} \mid ls,\ x_{\text{final}}.\mathsf{lock} \mid ls,\ \mathrm{L}(e_{\text{final}})$
$\mathrm{L}(e)$ is the lock that protects e. $x.\mathsf{lock}$ is the lock field stored in the Java object x.

We define a minimum lock level as follows: $l_{\min} ::= \infty \mid cn.l \mid \mathrm{LUB}(cn_1.l_1\ ...\ cn_k.l_k)$;
where $\mathrm{LUB}(cn_1.l_1\ ...\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$

Note that L(x) and LUB(...) are not computed—they are expressions used as such for type checking. The lock level ∞ denotes that the thread currently holds no locks.

We define the type system using the following judgments. We present the typing rules for these judgments after that.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash defn$	$defn$ is a well-formed class
$P; E \vdash_{\text{owner}} o$	o is an owner
$P; E \vdash constr$	constraint $constr$ is satisfied
$P; E \vdash X \succeq Y$	effect X subsumes effect Y
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash wf$	typing environment E is well-formed
$P \vdash field \in c$	class c declares/inherits $field$
$P \vdash meth \in c$	class c declares/inherits $meth$
$P; E \vdash field\ init$	$field\ init$ is a well-formed field initializer
$P; E \vdash meth$	$meth$ is a well-formed method
$P \vdash_{\text{level}} cn.l$	$cn.l$ is a well-formed lock level
$P \vdash cn_1.l_1 < cn_2.l_2$	$cn_1.l_1$ is less than $cn_2.l_2$ in the partial order formed by lock levels
$P \vdash cn.l < l_{\min}$	$cn.l$ is less than $l_{\min}$ in the partial order formed by lock levels
$P; E \vdash \text{level}(e) = cn.l$	e is a final expression owned by world and the lock level of e is $cn.l$
$P; E \vdash \text{level}(e) < l_{\min}$	e is a final expression owned by world and the lock level of e is less than $l_{\min}$
$P; E \vdash \text{Lock}(e) = r$	r is the lock that protects the final expression e
$P; E \vdash_{\text{final}} e : t$	e is a final expression with type t
$P; E \vdash e : t$	expression e has type t
$P; E; ls; l_{\min} \vdash e : t$	expression e has type t and evaluating e will not create data races or deadlocks

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses(P)}\quad \mathit{ClassOnce(P)}\quad \mathit{FieldsOnce(P)}\quad \mathit{MethodsOncePerClass(P)}\quad \mathit{OverridesOK(P)}\quad \mathit{LockLevelsOK(P)} \\ P = defn_{1..n}\ e \quad P \vdash defn_i \quad P; \emptyset; \mathsf{thisThread}; \infty \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash defn}$

[CLASS]

$$\frac{\begin{array}{c} E = \text{final } cn\langle f_{1..n}\rangle\ \mathsf{this},\ \text{owner } f_{1..n},\ f_i \succeq f_1,\ constr* \\ P; E \vdash wf \quad P; E \vdash c' \quad P; E \vdash field_i \quad P; E \vdash meth_i \end{array}}{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c'\ \mathsf{where}\ constr*\ \{level*\ field*\ meth*\}}$$

$\boxed{P; E \vdash X \succeq Y}$

[$X \succeq Y$]

$$\frac{\begin{array}{c} X = x_{1..n} \quad Y = y_{1..m} \\ \forall_{j\in\{1..m\}}\ \exists_{i\in\{1..n\}}\ (P; E \vdash x_i \succeq y_j) \end{array}}{P; E \vdash (X \succeq Y)}$$

$\boxed{P; E \vdash constr}$

[CONSTR ENV]

$$\frac{E = E_1,\ constr,\ E_2}{P; E \vdash constr}$$

[OWNER $\succeq$]

$$\frac{P; E \vdash e : cn\langle o_{1..n}\rangle}{P; E \vdash (o_1 \succeq e)}$$

[WORLD $\succeq$]

$$\frac{\begin{array}{c} P; E \vdash_{\text{owner}} o \\ P \vdash_{\text{level}} cn.l \end{array}}{P; E \vdash (\mathsf{world}{:}cn.l \succeq o)}$$

[REFL $\succeq$]

$$\frac{P; E \vdash_{\text{owner}} o}{P; E \vdash (o \succeq o)}$$

[TRANS $\succeq$]

$$\frac{\begin{array}{c} P; E \vdash (o_3 \succeq o_2) \\ P; E \vdash (o_2 \succeq o_1) \end{array}}{P; E \vdash (o_3 \succeq o_1)}$$

$\boxed{P; E \vdash_{\text{owner}} o}$

[OWNER WORLD]

$$\frac{P \vdash_{\text{level}} cn.l}{P; E \vdash_{\text{owner}} \mathsf{world}{:}cn.l}$$

[OWNER THREAD]

$$\frac{}{\begin{array}{c} P; E \vdash_{\text{owner}} \mathsf{thisThread} \\ P; E \vdash_{\text{owner}} \mathsf{otherThread} \end{array}}$$

[OWNER FORMAL]

$$\frac{E = E_1,\ \text{owner } f,\ E_2}{P; E \vdash_{\text{owner}} f}$$

[OWNER THIS]

$$\frac{E = E_1,\ c\ \mathsf{this},\ E_2}{P; E \vdash_{\text{owner}} \mathsf{this}}$$

$\boxed{P;\, E \vdash wf}$

[ENV $\emptyset$]
$$\frac{}{P;\, \emptyset \vdash wf}$$

[ENV X]
$$\frac{\begin{array}{c} P;\, E \vdash t \\ x \notin \mathrm{Dom}(E) \\ P;\, E \vdash wf \end{array}}{P;\, E,\, [\mathrm{final}]_{\mathrm{opt}}\ t\ x \vdash wf}$$

[ENV OWNER]
$$\frac{\begin{array}{c} f \notin \mathrm{Dom}(E) \\ P;\, E \vdash wf \end{array}}{P;\, E,\, \mathrm{owner}\ f \vdash wf}$$

[ENV CONSTR]
$$\frac{\begin{array}{c} constr = (o' \succeq o) \vee constr = (o' \not\succeq o) \\ P;\, E \vdash wf \qquad P;\, E \vdash_{\mathrm{owner}} o,\, o' \\ E' = E,\, constr \\ \nexists_{x,y}\ (P;\, E' \vdash y \succeq x) \wedge (P;\, E' \vdash y \not\succeq x) \end{array}}{P;\, E,\, constr \vdash wf}$$

[ENV LOCKSCLAUSE]
$$\frac{\begin{array}{c} locksclause = \mathsf{locks}(\ldots\ [e]_{\mathrm{opt}}) \\ P;\, E \vdash wf \end{array}}{P;\, E,\, locksclause \vdash wf}$$

$\boxed{P;\, E \vdash t}$

[TYPE INT]
$$\frac{}{P;\, E \vdash \mathsf{int}}$$

[TYPE OBJECT]
$$\frac{P;\, E \vdash_{\mathrm{owner}} o}{P;\, E \vdash \mathsf{Object}\langle o \rangle}$$

[TYPE C]
$$\frac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n} \rangle \ldots\ \mathsf{where}\ constr{*}\ \ldots \\ P;\, E \vdash_{\mathrm{owner}} o_i \qquad P;\, E \vdash o_i \succeq o_1 \\ P;\, E \vdash constr\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E \vdash cn\langle o_{1..n} \rangle}$$

$\boxed{P;\, E \vdash t_1 <: t_2}$

[SUBTYPE C]
$$\frac{\begin{array}{c} P;\, E \vdash cn\langle o_{1..n} \rangle \\ P \vdash \mathsf{class}\ cn\langle f_{1..n} \rangle\ \mathsf{extends}\ cn'\langle f_1\ o{*} \rangle\ \ldots \end{array}}{P;\, E \vdash cn\langle o_{1..n} \rangle <: cn'\langle f_1\ o{*} \rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]
$$\frac{\begin{array}{c} P;\, E \vdash t_1 <: t_2 \\ P;\, E \vdash t_2 <: t_3 \end{array}}{P;\, E \vdash t_1 <: t_3}$$

[SUBTYPE REFL]
$$\frac{P;\, E \vdash t}{P;\, E \vdash t <: t}$$

$\boxed{P;\, E \vdash \mathrm{level}(e) = cn.l}$

[LEVEL(EXP)]
$$\frac{P;\, E \vdash_{\mathrm{final}} e : cn'\langle \mathsf{world}{:}cn.l\ \ldots \rangle}{P;\, E \vdash \mathrm{level}(e) = cn.l}$$

$\boxed{P \vdash_{\mathrm{level}} cn.l}$

[LEVEL]
$$\frac{P \vdash \mathsf{class}\ cn\ldots\ \{\ldots\ \mathrm{LockLevel}\ l\ \ldots\}}{P \vdash_{\mathrm{level}} cn.l}$$

$\boxed{P;\, E \vdash \mathrm{level}(e) < l_{\min}}$

[LEVEL < LEVEL MIN]
$$\frac{\begin{array}{c} P;\, E \vdash \mathrm{level}(e) = cn.l \\ P \vdash cn.l < l_{\min} \end{array}}{P;\, E \vdash \mathrm{level}(e) < l_{\min}}$$

$\boxed{P \vdash cn_1.l_1 < cn_2.l_2}$

[LEVEL <]
$$\frac{P \vdash \mathsf{class}\ cn_1\ldots\ \{\ldots\ \mathrm{LockLevel}\ l_1 < \ldots\ cn_2.l_2\ \ldots\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

[LEVEL >]
$$\frac{P \vdash \mathsf{class}\ cn_2\ldots\ \{\ldots\ \mathrm{LockLevel}\ l_2 > \ldots\ cn_1.l_1\ \ldots\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

$\boxed{P \vdash cn.l < l_{\min}}$

[LEVEL < INFTY]
$$\frac{\begin{array}{c} l_{\min} = \infty \\ P \vdash_{\mathrm{level}} cn.l \end{array}}{P \vdash cn.l < l_{\min}}$$

[LEVEL < LUB]
$$\frac{\begin{array}{c} l_{\min} = \mathrm{LUB}(\ldots\ cn.l\ \ldots) \\ P \vdash_{\mathrm{level}} cn.l \end{array}}{P \vdash cn.l < l_{\min}}$$

[LEVEL < CN.L]
$$\frac{\begin{array}{c} l_{\min} = cn'.l' \\ P \vdash cn.l < cn'.l' \end{array}}{P \vdash cn.l < l_{\min}}$$

[LEVEL TRANS]
$$\frac{\begin{array}{c} P \vdash cn'.l' < l_{\min} \\ P \vdash cn.l < cn'.l' \end{array}}{P \vdash cn.l < l_{\min}}$$

$\boxed{P;\, E \vdash \mathrm{Lock}(e) = r}$

[LOCK THISTHREAD]
$$\frac{P;\, E \vdash e : cn\langle \mathsf{thisThread}\ o{*} \rangle}{P;\, E \vdash \mathrm{Lock}(e) = \mathsf{thisThread}}$$

[LOCK OTHERTHREAD]
$$\frac{P;\, E \vdash e : cn\langle \mathsf{otherThread}\ o{*} \rangle}{P;\, E \vdash \mathrm{Lock}(e) = \mathsf{otherThread}}$$

[LOCK WORLD]
$$\frac{P;\, E \vdash e : cn\langle \mathsf{world}{:}cn'.l'\ o{*} \rangle}{P;\, E \vdash \mathrm{Lock}(e) = e.\mathsf{lock}}$$

[LOCK FORMAL]
$$\frac{P;\, E \vdash e : cn\langle o_{1..n} \rangle \qquad E = E_1,\, \mathrm{owner}\ o_1,\, E_2}{P;\, E \vdash \mathrm{Lock}(e) = \mathrm{L}(e)}$$

[LOCK THIS]
$$\frac{P;\, E \vdash e : cn\langle \mathsf{this}\ o_{2..n} \rangle}{P;\, E \vdash \mathrm{Lock}(e) = \mathrm{Lock}(\mathsf{this})}$$

$\boxed{P;\, E \vdash \mathit{field\ init}}$

[FIELD INIT]
$$\frac{P;\, E;\, \mathsf{thisThread};\, \infty \vdash e : t}{P;\, E \vdash [\mathrm{final}]_{\mathrm{opt}}\ t\ fd = e}$$

$\boxed{P \vdash \mathit{field} \in c}$

[FIELD DECLARED]
$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n} \rangle \ldots\ \{\ldots\ \mathit{field}\ \ldots\}}{P \vdash \mathit{field} \in cn\langle f_{1..n} \rangle}$$

[FIELD INHERITED]
$$\frac{\begin{array}{c} P \vdash \mathit{field} \in cn\langle f_{1..n} \rangle \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m} \rangle\ \mathsf{extends}\ cn\langle o_{1..n} \rangle \ldots \end{array}}{P \vdash \mathit{field}\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m} \rangle}$$

$\boxed{P;\, E \vdash method}$

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ \text{owner}\ f_{1..n},\ constr*,\ arg_{1..n},\ \mathsf{locks}(\ldots\ [x]_{\text{opt}}) \qquad P;\, E' \vdash wf \qquad P;\, E' \vdash_{\mathsf{final}} x : t' \\ \forall_{i \in \{1..r\}}\ (P;\, E' \vdash_{\mathsf{final}} x_i : t_i \quad \wedge \quad P;\, E' \vdash \text{Lock}(x_i) = r_i) \\ P;\, E';\, \mathsf{thisThread},\ r_{1..r};\ \text{LUB}(cn_j.l_j\ ^{j \in 1..k}) \vdash e : t \end{array}}{P;\, E \vdash t\ mn\langle f_{1..n}\rangle(arg_{1..n})\ \mathsf{requires}(x_{1..r})\ \mathsf{locks}(cn_j.l_j\ ^{j\in 1..k}\ [x]_{\text{opt}})\ \mathsf{where}\ constr*\ \{e\}}$$

$\boxed{P \vdash meth \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle \ldots\ \{\ldots\ meth\ \ldots\}}{P \vdash meth \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{\begin{array}{c} P \vdash meth \in cn\langle f_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m}\rangle\ \mathsf{extends}\ cn\langle o_{1..n}\rangle \ldots \end{array}}{P \vdash meth\ [o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

$\boxed{P;\, E \vdash e : t}$

[EXP TYPE]

$$\frac{\exists_{ls}\ P;\, E;\, ls;\, \infty \vdash e : t}{P;\, E \vdash e : t}$$

$\boxed{P;\, E \vdash_{\text{final}} e}$

[FINAL VAR]

$$\frac{\begin{array}{c} P;\, E \vdash wf \\ E = E_1,\ \text{final}\ t\ x,\ E_2 \end{array}}{P;\, E \vdash_{\text{final}} x : t}$$

$\boxed{P;\, E;\, ls;\, l_{\min} \vdash e : t}$

[EXP SUB]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash e : t' \\ P;\, E;\, ls;\, l_{\min} \vdash t' <: t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash e : t}$$

[EXP NEW]

$$\frac{P;\, E \vdash c}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{new}\ c : c}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash x : t \\ P;\, E;\, ls;\, l_{\min} \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash x = e : t}$$

[EXP VAR]

$$\frac{E = E_1,\ t\ x,\ E_2}{P;\, E;\, ls;\, l_{\min} \vdash x : t}$$

[EXP LET]

$$\frac{\begin{array}{c} arg = [\text{final}]_{\text{opt}}\ t\ x \\ P;\, E;\, ls;\, l_{\min} \vdash e : t \\ P;\, E,\, arg;\, ls;\, l_{\min} \vdash e' : t' \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{let}\ (arg = e)\ \mathsf{in}\ \{e'\} : t'}$$

[EXP FORK]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash x_i : t_i \\ g_i = \text{final}\ t_i[\text{otherThread}/\text{thisThread}]\ x_i \\ P;\, g_{1..n};\, \mathsf{thisThread};\, \infty \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{fork}\ (x_{1..n})\ \{e\} : \mathsf{int}}$$

[EXP SYNC]

$$\frac{\begin{array}{c} P;\, E \vdash \text{level}(x) = cn.l < l_{\min} \\ (E = E_1,\ \mathsf{locks}(\ldots\ x'),\ E_2) \Longrightarrow (P;\, E \vdash cn.l < \text{level}(x')) \vee (x' = x) \\ P;\, E;\, ls,\, x;\, cn.l \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{synchronized}\ x\ \mathsf{in}\ e : t}$$

[EXP SYNC REDUNDANT]

$$\frac{\begin{array}{c} x \in ls \\ P;\, E;\, ls;\, l_{\min} \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash \mathsf{synchronized}\ x\ \mathsf{in}\ e : t}$$

[EXP REF]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\, E \vdash \text{Lock}(e) = r \\ (P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \wedge (r \in ls)) \vee (P \vdash (\mathsf{final}\ t\ fd) \in cn\langle f_{1..n}\rangle) \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash e.fd : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{\begin{array}{c} P;\, E;\, ls;\, l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\, E \vdash \text{Lock}(e) = r \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \wedge (r \in ls) \\ P;\, E;\, ls;\, l_{\min} \vdash e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n] \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash e.fd = e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} \text{Renamed}(\alpha) \stackrel{\text{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_m/f_m][e_1/y_1]..[e_k/y_k] \\ P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(x*)\ \mathsf{locks}(cn.l*\ [x']_{\text{opt}})\ \mathsf{where}\ constr*\ \ldots) \in cn\langle f_{1..n}\rangle \\ P;\, E;\, ls;\, l_{\min} \vdash e : cn\langle o_{1..n}\rangle \qquad P;\, E \vdash o_i \succeq o_1 \qquad P;\, E \vdash \text{Renamed}(constr) \\ P;\, E;\, ls;\, l_{\min} \vdash e_j : \text{Renamed}(t_j) \qquad P;\, E \vdash \text{Lock}(\text{Renamed}(x_i)) = r_i \qquad r_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \qquad x'' = \text{Renamed}(x') \qquad P;\, E \vdash (\text{level}(x'') < l_{\min}) \vee (\text{level}(x'') = l_{\min}) \wedge (x'' \in ls) \end{array}}{P;\, E;\, ls;\, l_{\min} \vdash e.mn\langle o_{(n+1)..m}\rangle(e_{1..k}) : \text{Renamed}(t)}$$

Chapter 5

Enabling Safe Software Upgrades in Persistent Object Stores

This chapter describes how SafeJava enables software upgrades in persistent object stores to be defined modularly and implemented efficiently.

Persistent object stores provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. Providing a satisfactory way of upgrading objects in a persistent object store has been a long-standing challenge. A natural way to define upgrades is for programmers to provide a *transform function* [138] for each class whose objects need to be upgraded. A transform function initializes the new form of an object using its current state. The system carries out the upgrade by using the transform functions to transform all objects whose classes are being replaced.

This way of handling upgrades introduces two problems:

1. The system must provide good semantics that let programmers reason about their transform functions locally, thus making it easy to design correct upgrades.
2. The system must run upgrades efficiently, both in space and time.

We provide solutions to both problems.

We first introduce a set of *upgrade modularity conditions* that constrain the behavior of an upgrade system. Any upgrade system that satisfies the conditions guarantees that when a transform function runs, it encounters only object interfaces and invariants that existed when its upgrade was defined. The conditions thus allow transform functions to be defined *modularly*: a transform function can be considered an extra method of the class being replaced, and can be reasoned about like the rest of the class. This is a natural assumption that programmers would implicitly make in any upgrade system—our conditions provide a grounding for this assumption. This way an upgrade system provides good semantics to programmers who design upgrades.

We then show how upgrades implemented in SafeJava can execute efficiently, while satisfying the upgrade modularity conditions. Previous approaches do not provide a satisfactory solution to this problem. An upgrade system could satisfy the conditions by keeping old versions of all objects, since old versions preserve old interfaces and old object states. However versions are expensive, and to be practical, an upgrade system must avoid them most of the time. Some earlier systems [116, 13, 100] avoid versions by severely limiting the expressive power of upgrades (e.g., transform functions are not allowed to make method calls); others [8, 114] limit the number of versions using a stop-the-world approach that shuts down the system for upgrade and discards the versions when the upgrade is complete; yet others [138] do not satisfy the upgrade modularity conditions that enable programmers to reason about their upgrades locally.

Our approach provides an efficient solution to this problem. We perform upgrades *lazily*; we don't prevent application access to persistent objects by stopping the world but instead transform objects just before they are accessed by an application. We do this without requiring the use of versions most of the time. Also, we impose no limitations on the expressive power of transform functions. Yet we provide good semantics: our upgrade system satisfies the upgrade modularity conditions and thus supports local reasoning.

Our approach exploits the fact most transform functions are *well behaved*: they access only the object being transformed and its encapsulated objects. SafeJava statically checks if transform functions are well behaved. If they are, the runtime system provides an efficient way to enforce the upgrade modularity conditions without maintaining versions. If they aren't, we provide an additional mechanism, *triggers*, which can be used to control the order of transform functions to satisfy the conditions. If even triggers are insufficient, we use versions but only in cases where they are needed.

The rest of this chapter is organized as follows. Section 5.1 presents our upgrade modularity conditions. Section 5.2 describes how our system executes upgrades. Section 5.3 shows that our system satisfies the upgrade modularity conditions. Section 5.4 presents related work, and Section 5.5 concludes.

5.1 Semantics of Upgrades

This section describes the upgrade model. It also defines the upgrade modularity conditions and explains why they make it easy for programmers to reason about upgrades.

5.1.1 System Model

We assume a persistent object store (e.g., an object-oriented database) that contains conventional objects similar to what one might find in an object-oriented programming language such as Java. Objects refer to one another and interact by calling one another's methods. The objects belong to classes that define their representation and methods. Each class implements a type. Types are arranged in a hierarchy. A type can be a subtype of one or more types. A class that implements a type implements all supertypes of that type.

We assume that applications access persistent objects within atomic transactions, since this is necessary to ensure consistency for the stored objects; transactions allow for concurrent access and they mask failures. An application transaction consists of calls on methods of persistent objects as well as local computation. A transaction terminates by committing or aborting. If the commit succeeds, changes become persistent. If instead the transaction aborts, none of its changes affect the persistent objects.

Upgrades in a persistent object store can be used to improve an object's implementation, to make it run faster, or to correct an error; to extend the object's interface, e.g., by providing it with additional methods; or even to change the object's interface in an *incompatible* way, so that the object no longer behaves as it used to, e.g., by removing one of its methods or redefining what a method does. Incompatible upgrades are probably not common but they can be important in the face of changing application requirements.

5.1.2 Defining Upgrades

An upgrade is defined by describing what should happen to classes that need to be changed. The information for a class that is changing is captured in a *class-upgrade*. A class-upgrade is a tuple:

$$\langle \text{old-class, new-class, TF} \rangle$$

A class-upgrade indicates that all objects belonging to old-class should be transformed, through use of the *transform function*, TF, into objects of new-class. TF takes an old-class object and a newly allocated new-class object and initializes the new-class object from the old-class object. The upgrade system causes the new-class object to take over the identity of the old-class object, so that all objects that used to refer to the old-class object now refer to the new-class object.

This mechanism preserves object state and identity. The preservation is crucial, because the whole point of a persistent store is to maintain object state. When objects are upgraded, their state must survive, albeit in a modified form as needed in the new class. Furthermore, a great deal of object state is captured in the web of object relationships. This information is expressed by having objects refer to other objects. When an object is upgraded it must retain its identity so that objects that referred to it prior to the upgrade still refer to it.

An upgrade is a set of one or more class-upgrades. When an upgrade changes the interface of a class C *incompatibly*, so that its objects no longer behave as they used to, this may affect other classes, including subclasses of C and classes that use types C no longer implements. All affected classes have to be upgraded as well, so that the new system as a whole remains type correct. A *complete upgrade* contains class-upgrades for all classes that need to change due to some class-upgrade already in the upgrade [8, 51, 60, 138]. Completeness is checked using rules analogous to type checking.

Our system accepts an upgrade only if it is complete. At this point we say the upgrade is *installed*. Once an upgrade has been installed, it is ready to run. An upgrade is executed by running transform functions on all affected objects, i.e., all objects belonging to old classes.

5.1.3 Upgrade Modularity Conditions

As we mentioned in the introduction of this chapter, an upgrade system must guarantee that when a transform function runs, it encounters only interfaces that existed at the time its upgrade was installed and states that satisfy its object's invariants. This guarantee means the transform function writer need not be concerned, when reasoning about correctness of upgrades, with object interfaces and object invariants that existed in the past or will exist in the future. Instead, the transform function can be thought of as an extra method of the old-class: the writer can assume the same invariants and interfaces as are assumed for the other methods.

The job of the upgrade system is to run upgrades in a way that supports this modularity property. There are two different problems that must be solved. First is the question of how to order upgrades relative to application transactions and to other upgrades. Second is the issue of how to order the transform functions belonging to a single upgrade.

Ordering Upgrades

The requirement for ordering of entire upgrades is simple: upgrades are transactions and thus must be serialized relative to application transactions and to one another: a later upgrade must appear to run after an earlier one.

An upgrade transaction transforms all objects of old-classes. We view each transform function as running in its own transaction; each such *transform transaction* is the execution of a transform function on one object. The entire upgrade transaction thus consists of the execution of all the transform transactions for that upgrade.

Now we can state the serializability requirement. A similar condition is given in [138]. We use the notation $[A1; A2]$ to mean that $A1$ ran before $A2$.

M1. If we have $[A;\ \mathrm{TF}(x)]$, where A is either an application transaction that is serialized after TF's upgrade is installed, or A is a transform function from a later upgrade, this has the same effect as $[\mathrm{TF}(x);\ A]$.

An upgrade system that stops the world to run an upgrade transaction and only allows application transactions to continue after that transaction completes (e.g., [8, 114]) satisfies this condition trivially, since the order [A; TF(x)] won't occur for either later application transactions or later upgrades. An upgrade system that doesn't stop the world will have to ensure that when it runs A before some transform that must be serialized before A, the effect will be the same as if they ran in the opposite order.

Order within Upgrades

Condition M1 says nothing about how the upgrade system chooses the ordering of transforms within an upgrade. The following two conditions constrain this order.

M2. If TF(x) and TF(y) are from the same upgrade and TF(x) (transitively) uses y and we have $[\mathrm{TF}(y);\mathrm{TF}(x)]$, this has the same effect as $[\mathrm{TF}(x);\ \mathrm{TF}(y)]$.

M3. If TF(x) and TF(y) are from the same upgrade and TF(x) does not (transitively) use y and TF(y) does not (transitively) use x, then $[\mathrm{TF}(y);\ \mathrm{TF}(x)]$ has the same effect as $[\mathrm{TF}(x);\ \mathrm{TF}(y)]$.

Here TF(x) *uses* y if TF(x) reads/writes a field of y or calls a method of y. *Transitively uses* means that this action may occur via uses of intermediate objects, e.g., TF(x) calls a method of z, which calls a method of y.

Condition M2 states that if transform function TF(x) uses object y, the behavior of the system must be the same as if TF(x) ran before TF(y). Condition M3 states that for unrelated objects, the behavior of the system must be independent of the order in which their transforms ran; the upgrade system can then choose any order for the two transforms.

Some upgrade systems satisfy Condition M2 by using versions (so that when TF(x) runs it sees the old version of y); this is the approach taken in [8, 114]. Others avoid the problem altogether by limiting the expressive power of transforms so that they cannot make method calls, as in [116, 13, 100]. A third possibility (and the direction we follow) is to satisfy M2 by controlling the order of transforms so that when TF(x) uses y, x is transformed before y.

Conditions M1-M3 together ensure upgrade modularity: transform functions encounter the expected interfaces and object invariants because upgrades run in upgrade order, application transactions do not interfere with transform functions, transform functions of unrelated objects do not interfere with each other, and transform functions of related objects appear to run in a pre-determined order (namely an object appears to be transformed before its subobjects). Thus these conditions allow transform functions to be reasoned about locally, as extra methods of old classes. Writers of transform functions can assume the same invariants and interfaces as are assumed for the other methods of old classes.

5.2 Executing Upgrades

This section describes our lazy upgrade system.

Stopping the world to run an upgrade is undesirable since it can make the system unavailable to applications for a long time. Our system avoids delaying applications by running the upgrade incrementally just in time.

The system runs each transform function as an individual transaction. These transactions are interleaved with application transactions. When an application transaction A is about to use an object that is due to be transformed, the system interrupts A and runs the transform function at that point; this way we ensure that application transactions never observe untransformed objects.

The transform transaction T must be serialized *before* A in the commit order since A uses the transformed object initialized by T. Therefore, if T reads or modifies an object modified by A or if T modifies an object read by A, the system aborts A. A is highly unlikely to abort, however; we discuss how our techniques avoid having to abort A in Sections 5.3.2 and 5.3.3.

When T finishes executing, it commits. Then the system continues running A, or if A was aborted, it reruns A.

Our system does not require that an earlier upgrade complete before a later upgrade starts since this might delay the later upgrade for a long time. Instead many upgrades can be in progress at once and several transforms may be pending for an object. The system runs pending transforms for an object in upgrade order. While running transform transaction T, the system might encounter an object that has pending transforms from upgrades earlier than the one that defined T; in this case, the system interrupts T (just as it interrupted A) to run the pending transforms.

5.3 Enforcing Upgrade Modularity Conditions

The lazy approach described in Section 5.2 ensures that transforms of individual objects run in upgrade order and that applications running after an upgrade never observe objects that need to be transformed. But it does not enforce the upgrade modularity conditions discussed in Section 5.1.3, and thus it does not provide the desired semantics that allow programmers to reason locally about their transform functions. For example, it's possible that application transaction $A1$ uses object x causing it to be transformed and changing its interface incompatibly; later $A2$ uses y and when TF(y) runs it uses x and encounters the unexpected interface.

For the system to provide good semantics, we must prevent this kind of occurrence. Our approach is based on object encapsulation. This section shows how upgrades implemented in SafeJava can execute efficiently without needing versions, while satisfying the upgrade modularity conditions.

5.3.1 Object Encapsulation and Upgrades

Object encapsulation enables local reasoning about program correctness in object-oriented programs. Object encapsulation also facilitates modular upgrades in our system because it imposes an order on transforms. If y is encapsulated within x, applications must access x before y and therefore x will be transformed before y. This means that when TF(x) runs it will see the proper interface for y.

However, if TF(x) accesses some object z referred to by x (directly or indirectly) but not encapsulated within x, it might encounter an unexpected interface or state. Our system provides good semantics without using versions for transforms that don't do this. Such transforms satisfy Condition E:

E. TF(x) only uses x and objects that x encapsulates.

We say such transforms are *well behaved*.

Condition E can be statically checked by the SafeJava compiler. We presented the SafeJava type system for enforcing object encapsulation and checking side effects of methods in Chapter 2. The same type system that we presented in Section 3.A at the end of Chapter 3 can also check Condition E.

5.3.2 Ensuring Upgrade Modularity

This section shows that SafeJava can ensure Conditions M1-M3, assuming Condition E holds for all TFs.

For any object x affected by an upgrade, our system guarantees that x is accessed before any object encapsulated within x. Thus the system ensures the following conditions:

S1. TF(x) runs before A uses x or any object encapsulated within x, where A is either an application transaction that ran after TF's upgrade was installed, or A is a transform function from a later upgrade.

S2. If TF(x) and TF(y) are in the same upgrade and y is encapsulated within x, then TF(x) runs before TF(y).

Note that our system handles inner classes specially to ensure Condition S1 and S2. When a upgrade affects a class, we attach triggers to its inner classes; this is done automatically as part of installing the upgrade. Then when an inner class object is used, the trigger causes the outer object to be transformed.

Now we give informal proofs that when E holds, S1 and S2 ensure that Conditions M1-M3 hold. Our proofs consider only adjacent transactions, but this is sufficient because M1-M3 can be used to reorder sequences containing intervening transactions to achieve adjacency.

M1: If we have [A; TF(x)], where A is either an application transaction that is serialized after TF's upgrade is installed, or A is a transform function from a later upgrade, this has the same effect as [TF(x); A].

Proof: Since A ran before TF(x), we know from S1 that A does not use x or any object x encapsulates. Furthermore, we know from E that TF(x) only uses x and objects x encapsulates. Therefore the read/write sets of A and TF(x) have no object in common and thus the effect is the same as if TF(x) ran before A. □

M2: If TF(x) and TF(y) are from the same upgrade and TF(x) (transitively) uses y and we have [TF(y);TF(x)], this has the same effect as [TF(x); TF(y)].

Proof: Since TF(x) (transitively) uses y, we know from E that x encapsulates y. Therefore, we know from S2 that TF(x) runs before TF(y). Thus the condition holds trivially because the order [TF(y); TF(x)] will not occur. □

M3: If TF(x) and TF(y) are from the same upgrade and TF(x) does not (transitively) use y and TF(y) does not (transitively) use x, then [TF(x); TF(y)] is equivalent to [TF(y); TF(x)].

Proof: TF(x) and TF(y) can commute unless there is some object z that is read by one TF and modified by the other. If such an object exists, we know from E that both x and y must encapsulate it. Therefore the existence of z implies that either y is encapsulated within x and z is encapsulated within y, or x is encapsulated within y and z is encapsulated within x. But we know from S2 that an encapsulating object is used before any object it encapsulates. Therefore whichever object encapsulates the other, the TF for that object must use the other before using z, which violates the assumption that neither TF uses the other object. □

When E holds we also get another benefit. Recall from Section 5.2 that our system will abort an interrupted transaction if it has a read/write conflict with a TF. However, when E holds there will be no conflicts. This is because the interrupted transaction cannot use any object that a pending transform function will use without first causing that pending transform function to run.

5.3.3 Triggers and Versions

Now we consider what happens when a TF violates Condition E. Condition E states that TF(x) can only use x and objects encapsulated within x. There are two reasons why the condition might not hold.

Violations of Condition E

The first reason is that a TF(x) might use objects that x does not depend on (directly or transitively). For example, the depends-on relation in Section 5.3.1 is intentionally limited to not include immutable subobjects, since correctness does not require encapsulation of such subobjects. However, if the subobjects are no longer immutable after an upgrade and if a transform function reads such subobjects, Condition E would be violated. But such upgrades are unlikely to happen in practice.

The second reason Condition E may not hold is that an object might not encapsulate subobjects it depends on. This might occur with cyclic objects. It also might occur in the case of iterators [104, 71] and other similar constructs.

Consider, for example, an iterator over a set s. The iterator's job is to return a different element of the set each time its `next` method is called until all elements of the set have been returned. To do this job efficiently, the iterator needs direct access to the objects that represent s, e.g., if s is implemented using a linked list, the iterator must be able to access the nodes in the linked list directly. But the iterator cannot be encapsulated within s because we would like it to be used by objects outside s.

To allow efficient implementation of iterators, a set object does not encapsulate the linked list, even though it depends on it. This is because the iterator is an outside object that can access the list. In fact, what is really happening with iterators is that more than one object depends on some shared subobject. For example, both s and iterators over s depend on the linked list.

Encapsulation violations of this sort do not prevent local reasoning in object-oriented programs, so long as all the code with the shared dependencies is in the same module. If the code is modularized like this, correctness can still be reasoned about locally, by considering the module as a whole. For example, as we explained in Chapter 2, the iterator could be implemented as an inner class of the set class, and modular reasoning would still be possible [24]. However, such encapsulation violations can lead to a violation of E.

Handling Violations of Condition E

When E is violated there are two possible solutions: explicitly order the transform functions so that Conditions M1-M3 are not violated, or use versions. Since the decision about which approach to use requires an understanding of program behavior, the programmer must instruct the system about what to do.

Explicit ordering of transform functions is possible when x and all the unencapsulated objects used by TF(x) are encapsulated within a containing object. For example, suppose the linked list class is being upgraded incompatibly, and as a result a set and all its iterators must also be transformed. If a containing object encapsulates both the set object and its iterator objects, we can force the set and the iterators to be transformed before the linked list by attaching a *trigger* to the class of the containing object.

A trigger is a function that takes an object as an argument and returns a list of objects needing to be upgraded. Triggers are defined as part of an upgrade (in addition to the class-upgrades); such a definition identifies the class being triggered and provides the code for the trigger. The system runs the trigger when an object of the class is first used (after the upgrade is installed); then it processes the list (in list order) and runs any pending transform functions on the objects in the list. SafeJava statically checks that the trigger on an object x uses only objects that x depends on and furthermore to only reads those objects. The uses restriction ensures that the trigger itself can be reasoned about modularly; the read-only restriction guarantees that the trigger cannot affect system state. Given these restrictions, triggers provide M1-M3 because they control order: they provide M1 and M2 because [A; TF(x)] and [TF(y); TF(x)] cannot occur.

When there is no containing object, or when there is no way to ensure a correct order for transforms (e.g., because a group of objects with cyclic dependencies is being transformed), we have to fall back on versions. In this case, we keep old versions for any unencapsulated object used by the offending transform function TF(x); for each such object z, we also keep versions for all objects it depends on. SafeJava statically enforces the restriction that transform functions do not modify old versions of objects. Given this restriction on transform functions, versions provide M1-M3, because immutable versions preserve the old interfaces and object states.

Triggers and versions also ensure that a transform does not conflict with interrupted transactions. Furthermore, the system can interact with the user prior to installing an upgrade to help the user include needed triggers and versions. Therefore our system makes it highly unlikely that running a transform will cause interrupted transactions to abort.

Implementation

This section discusses implementation issues for supporting upgrades. We only describe the general strategy here. The complete details of how the implementation works within the Thor object-oriented database can be found in [25].

Thor is a client-server system. Persistent objects reside at servers; application transactions run at client machines on cached copies of persistent objects. Thor uses optimistic concurrency control [1]. Client machines track objects used and modified by a transaction. When a transaction attempts to commit, the client sends a commit request containing information about used objects and states of new and modified objects to one of the servers. The server decides whether the transaction can commit (using two-phase commit if the transaction used objects at more than one server) and informs the client of its decision. More information about Thor can be found in [102, 33, 1, 20].

Installing Upgrades

Upgrades are installed by interacting with one of the servers. This server checks the upgrade for completeness. It interacts with the user to determine whether Condition E holds; if it doesn't, this may result in a trigger or version being added to the upgrade. If versions are needed, these are also described by class-upgrades, marked as requiring versions. When all needed information has been added to the upgrade, the server notifies clients and other servers about the new upgrade.

Information about transforms, triggers, and versions is attached to class objects of old-classes. E.g., the old-class object points to the transform function.

Running Upgrades

As mentioned, we interrupt application transactions and transform transactions when we encounter objects that need to be upgraded or have triggers attached to them. This processing involves the following steps:

1. Each time an application transaction, AT, or a transform transaction, TT, uses an object, we check whether that object needs to be transformed or has an attached trigger. If so, we interrupt AT or TT and start a transaction T to run the transform code on that object. This step insures that application code encounters only fully upgraded objects, and pending transforms encounter objects of expected versions.

2. We run transaction T. If T conflicts with an interrupted transaction (reads or modifies a modified object or modifies a read object) we abort all the interrupted transactions including AT.

3. When T completes, we create a version for objects it modifies, if that is indicated.

4. If T has triggered some other transforms we run them provided they are defined by upgrades no later than the upgrade that caused T to run. Note that T is finished executing at this point; we don't interrupt it to run these additional transforms.

5. When there are no triggered transforms left to run, we continue running the interrupted AT or TT, unless these were aborted.

6. When processing is complete (either because the AT is ready to commit, or because the AT was forced to abort), we commit all completed transactions in their completion order. If some transaction's commit fails, we abort those that haven't committed yet and then rerun them in the same order as before. Then we rerun the application transaction if it aborted.

Implementation in Thor and Performance Results

Details on the Thor upgrade implementation can be found in [25]. The paper also sketches an alternative approach for implementing upgrades that can be used in other persistent object systems.

The paper also presents results of a performance study indicating the upgrade infrastructure has low cost. It has negligible impact (less than 1% overhead) on applications that do not use objects that need to be upgraded. We expect this to be the common case because upgrades are likely to be rare (e.g., once a week or once a day). The results also show that when upgrades are needed, the overhead of transforming an object is small.

5.4 Related Work

There has been much research on software upgrades and data transformation covering a broad range of research topics. The work on schema or class versioning (e.g., [49, 125, 40]) considers multiple co-existing versions of a schema or class. The work on object instance evolution (e.g., [14, 61]) considers selective transformation of some but not all objects in a class. The work on hot-swapping of modules (e.g., [83, 62, 85]) is concerned with updating a class while there is executing code that is using objects of the class; this work considers issues of type safe access to the same object via multiple potentially incompatible interfaces but does not enforce the upgrade modularity conditions that allow programmers to reason locally about the correctness of their upgrades.

Here we focus on work on schema evolution in persistent object stores (such as object-oriented databases), since this is the work most closely related to our own. In these systems the database has one logical schema to which modifications of class definitions are applied; all object instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. The schema evolution approach is used in Orion [13], OTGEN [100], O2 [58, 138], GemStone [30, 116], Objectivity/DB [115], Versant [130], and PJama [9, 8] systems, and is the only approach available in commercial RDBMS. An extensive survey of the previous schema evolution systems can be found in [60].

None of the previous schema evolution systems provide a way of executing upgrades efficiently both in space and time, while allowing programmers to reason locally about the correctness of their upgrades. To be practical, systems must avoid keeping old versions of objects most of the times. Some earlier systems [116, 13, 100] avoid versions by severely limiting the expressive power of upgrades (e.g., transform functions are not allowed to make method calls). Others [8, 114] limit the number of versions using a stop-the-world approach that shuts down the system for upgrade and discards the versions when the upgrade is complete

Very few systems support lazy conversion and *complex* (fully expressive) transforms. The work on O2 [58, 138] was the first to identify the problem posed by deferred complex transforms and incompatible upgrades. This work introduced an upgrade modularity condition that is based on the equivalence of lazy and eager conversion. This condition is weaker than our Conditions M1-M3 because it does not consider the interleavings of transforms from the same upgrade.

O2 ensures type safety for deferred complex transforms using a "screening" approach similar to versioning. Unlike our approach, however, analysis in O2 does not take encapsulation into account. When an incompatible upgrade occurs after a complex transform is installed, O2 either activates an eager conversion or avoids transform interference by keeping versions for all objects. This approach is unnecessarily conservative (it switches to eager execution even when E holds). Also, O2 does not solve the problem of applications modifying objects that are then used by transforms from earlier upgrades; this is unsafe because it violates Condition M1.

Implementation details for commercial systems supporting lazy conversion with complex transforms are generally not available. We found limited information for O2, e.g., we found no information about the mechanisms for supporting the atomicity of individual transforms, or about the performance impact of upgrade support on normal case operation. The O2 screening approach co-locates versions of upgraded objects physically near the new version of the object [64]. This requires database reorganization when versions are created. In contrast, our system does not require co-location of object versions; this allows us to preserve clustering of non-upgraded objects without database reorganization and furthermore, we are often able to preserve clustering for upgraded objects as well. Preserving clustering is important for system performance because of its impact on disk access [65].

Some implementation issues caused by complex user-defined transforms arise in eager as well as lazy systems, e.g., either has to support arbitrary order of transforms and access to potentially incompatible transformed objects. The PJama system [8, 60] keeps old and new versions to solve this problem. To provide recoverability and reduce memory demands when converting large datasets, it performs incremental partitioned conversion that creates partitions with old and new versions, and at the end of conversion deletes the old copies by copying the converted partitions. Like our system, PJama uses write-ahead logging to support conversion atomicity and recoverability.

5.5 Conclusions

Persistent object stores provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. Providing a satisfactory way of upgrading objects in a persistent object store has been a long-standing challenge. Upgrades must be performed in a way that is efficient both in space and time, and that does not stop application access to the store. In addition, however, the approach must be modular: it must allow programmers to reason locally about the correctness of their upgrades similar to the way they would reason about regular code. This chapter provides solutions to both problems.

This chapter defines *upgrade modularity conditions* that any upgrade system must satisfy to support local reasoning about upgrades. These conditions are more general than earlier definitions [138]: they apply to both lazy and stop-the-world upgrade systems; they also apply to both systems that use versions and systems that don't.

The chapter then shows how upgrades implemented in SafeJava can execute efficiently, while satisfying the upgrade modularity conditions. The approach exploits object encapsulation properties in a novel way. The chapter proves that our upgrade system satisfies the upgrade modularity conditions when transforms are well behaved. We also show that the conditions hold through the use of triggers and versions.

We have a prototype implementation that supports fully expressive, modular, lazy upgrades. The implementation is done in Thor [102, 20]. We also have an alternate implementation approach that can be used in any persistent object system.

The results of our performance study indicate that the infrastructure has low cost. It has negligible impact on applications that do not use objects that need to be upgraded. We expect this to be the common case because upgrades are likely to be rare (e.g., once a week or once a day). The results also show that when upgrades are needed, the overhead of transforming an object is small.

Our approach thus provides a complete solution to the problem of upgrading persistent objects.

Chapter 6

Enabling Safe Region-Based Memory Management

The Real-Time Specification for Java (RTSJ) [18] provides a framework for building real-time systems. The RTSJ allows a program to create real-time threads with hard real-time constraints. These real-time threads cannot use the garbage-collected heap because they cannot afford to be interrupted for unbounded amounts of time by the garbage collector. Instead, the RTSJ allows these threads to use objects allocated in immortal memory (which is never garbage collected) or in regions [128]. Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access heap references.

SafeJava introduces a new static type system for writing real-time programs in Java. SafeJava guarantees that the RTSJ runtime checks will never fail for well-typed programs. SafeJava thus serves as a front-end for the RTSJ platform. It offers two advantages to real-time programmers. First, it provides an important safety guarantee that a program will never fail because of a failed RTSJ runtime check. Second, it allows RTSJ implementations to remove the RTSJ runtime checks and eliminate the associated overhead.

Our approach is applicable even outside the RTSJ context; it could be adapted to provide safe region-based memory management for other real-time languages as well.

The SafeJava type system makes several important technical contributions over previous type systems for region-based memory management. For object-oriented programs, it combines region types [39, 48, 80, 128] and ownership types [23, 24, 26, 41, 43] in a unified type system framework. Region types statically ensure that programs never follow dangling references. Ownership types statically enforce object encapsulation and enable modular reasoning about program correctness in object-oriented programs. Consider, for example, a Stack object s that is implemented using a Vector object v. To reason locally about the correctness of the Stack implementation, a programmer must know that v is not directly accessed by objects outside s. With ownership types, one can declare that s *owns* v. The type system then statically ensures that v is encapsulated within s.

In an object-oriented language that only has region types (e.g., [39]), the types of s and v would declare that they are allocated in some region r. In an object-oriented language that only has ownership types, the type of v would declare that it is owned by s. SafeJava provides a simple unified mechanism to declare *both* properties. The type of s can declare that it is allocated in r and the type of v can declare that it is owned by s. SafeJava then statically ensures that both objects are allocated in r, that there are no pointers to v and s after r is deleted, and that v is encapsulated within s. SafeJava thus combines the benefits of region types and ownership types.

SafeJava extends region types to multithreaded programs by allowing explicit memory management for objects shared between threads. It allows threads to communicate through objects in *shared regions* in addition to the heap. A shared region is deleted when all threads exit the region. However, programs in a system with only shared regions (e.g., [79]) will have memory leaks if two long-lived threads communicate by creating objects in a shared region. This is because the objects will not be deleted until both threads exit the shared region. To solve this problem, SafeJava introduces *subregions* within a shared region. A subregion can be deleted more frequently, for example, after each loop iteration in the long-lived threads.

SafeJava introduces *typed portal fields* in subregions to serve as a starting point for inter-thread communication. Portals also allow typed communication, so threads do not have to downcast from Object to more specific types. SafeJava therefore avoids any dynamic type errors associated with these downcasts. SafeJava also introduces user-defined *region kinds* to support subregions and portal fields.

SafeJava extends region types to real-time programs by statically ensuring that real-time threads do not interfere with the garbage collector. SafeJava augments region kind declarations with *region policy* declarations. It supports two policies for creating regions as in the RTSJ. A region can be an LT (Linear Time) region, or a VT (Variable Time) region. Memory for an LT region is preallocated at region creation time, so allocating an object in an LT region only takes time proportional to the size of the object (because all the bytes have to be zeroed). Memory for a VT region is allocated on demand, so allocating an object in a VT region takes variable time. SafeJava checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions.

SafeJava also prevents an RTSJ *priority inversion* problem. In the RTSJ, any thread entering a region waits if there are threads exiting the region. If a regular thread exiting a region is suspended by the garbage collector, then a real-time thread entering the region might have to wait for an unbounded amount of time. SafeJava statically ensures that this priority inversion problem cannot happen.

Finally, we note that most previous region type systems allow programs to create, but not follow, dangling references. Such references can cause a safety problem when used with copying garbage collectors. SafeJava, on the other hand, prevents a program from creating dangling references in the first place.

Outline

This chapter extends the type system we presented so far to support safe region-based memory management. The rest of this chapter is organized as follows. Sections 6.1, 6.2, and 6.3 describe the SafeJava type system. Section 6.4 presents some of the important rules

SJ1. Every object has an owner.

SJ2. The owner can either be another object or a region.

SJ3. The ownership relation forms a forest of trees.

SJ4. The owner of an object does not change over time.
(Except if there is a unique pointer to that object.)

SJ5. If object z owns y but $z \not\succeq x$, then x cannot access y.
(Except if x is an inner class object of z.)

Figure 6-1: Ownership Properties

SJ8. Objects directly or transitively owned by a thread-local region are local to the corresponding thread. All other objects are potentially shared between threads.

SJ9. By default, every object is protected by the lock on the root owner of that object. ro is the root owner of an object o iff $ro \succeq o$ and some region directly owns ro.
An object can also be protected by an arbitrary lock.

SJ10. To access to an object, a thread must hold the lock that protects that object.
A thread can however access thread-local objects (that are owned directly or transitively by thread-local regions) without any synchronization.
Immutable objects and objects with unique pointers can also be accessed without synchronization.

Figure 6-2: Properties of Thread-Local and Shared Objects

for typechecking. The complete set of rules are in the appendix at the end of this chapter. Section 6.5 describes type inference techniques. Section 6.6 describes how programs written in SafeJava are translated to run on our RTSJ platform. Section 6.7 describes our experience in using SafeJava. Section 6.8 presents related work and Section 6.9 concludes.

6.1 Regions for Object-Oriented Programs

This section presents the SafeJava type system for safe region-based memory management in single-threaded object-oriented programs.

Ownership Relation

Objects in SafeJava are allocated in regions. As before, the key to the SafeJava type system is the concept of object ownership. Recall the ownership properties from Figure 4-1 and properties of thread-local and shared objects from Figure 4-2. In this chapter, we extend the properties as shown in Figures 6-1 and 6-2.

Every object has an owner. An object can be owned by another object, or by a region. As before, objects owned by another object are encapsulated within that object. Objects owned

SJ15. If region $r \succeq$ object x, then x is allocated in r.

SJ16. For any region r, heap $\succeq_r r$ and immortal $\succeq_r r$.

SJ17. $x \succeq y \Longrightarrow x \succeq_r y$.

SJ18. If region $r_1 \succeq$ object o_1, region $r_2 \succeq$ object o_2, and $r_2 \not\succeq_r r_1$, then o_1 cannot contain a pointer to o_2.

Figure 6-3: Properties of Regions

by a region are unencapsulated and correspond to objects owned by world or thisThread from previous chapters. Objects directly or transitively owned by a thread-local region are local to the corresponding thread. All other objects are potentially shared between threads.

SafeJava statically ensures the properties shown in Figure 6-3 for regions. Recall from Section 2.2 that we write $o_1 \succeq o_2$ if o_1 directly or transitively owns o_2 or if o_1 is the same as o_2. The relation $\succeq$ is thus the reflexive transitive closure of the *owns* relation. Property SJ15 states that if an object is owned by a region, then that object and all its subobjects are allocated in that region.

Figure 6-7 shows an example ownership relation. We draw a solid line from x to y if x owns y. Region r2 owns s1, s1 owns s1.head and s1.head.next, etc.

Outlives Relation

SafeJava allows programs to create regions. It also provides two special regions: the garbage collected region heap, and the "immortal" region immortal. The lifetime of a region is the time interval from when the region is created until it is deleted. If the lifetime of a region r_1 includes the lifetime of region r_2, we say that r_1 *outlives* r_2, and write $r_1 \succeq_r r_2$. The relation $\succeq_r$ is thus reflexive and transitive. We extend the *outlives* relation to include objects. We define that $x \succeq y$ implies $x \succeq_r y$. The extension is natural: if object o_1 owns object o_2 then o_1 outlives o_2 because o_2 is accessible only through o_1. Also, if region r owns object o then r outlives o because o is allocated in r.

Our outlives relation has the properties shown in Figure 6-3. SJ16 states that heap and immortal outlive all regions. SJ16 states that the outlives relation includes the ownership relation. SJ17 states our memory safety property, that if object o_1 in region r_1 contains a pointer to object o_2 in region r_2, then r_2 outlives r_1. SJ17 implies that there are no dangling references in SafeJava. Figure 6-7 shows an example outlives relation. We draw a dashed line from region x to region y if x outlives y. In the example, region r1 outlives region r2, and heap and immortal outlive all regions.

The following lemmas follow trivially from the above definitions:

LEMMA 3. *If object $o_1 \succeq_r$ object o_2, then $o_1 \succeq o_2$.*

LEMMA 4. *If region $r \succeq_r$ object o, then there exists a unique region r' such that $r \succeq_r r'$ and $r' \succeq o$.*

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ e \\
\mathit{defn} & ::= & \mathsf{class}\ cn\langle \mathit{formal}+\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ \mathit{constr}^*\ \{\mathit{field}^*\ \mathit{meth}^*\} \\
c & ::= & cn\langle \mathit{owner}+\rangle \mid \mathsf{Object}\langle \mathit{owner}\rangle \\
\mathit{owner} & ::= & fn \mid r \mid \mathsf{this} \mid \mathsf{initialRegion} \mid \mathsf{heap} \mid \mathsf{immortal} \\
\mathit{constr} & ::= & \mathit{owner}\ \mathsf{owns}\ \mathit{owner} \mid \mathit{owner}\ \mathsf{outlives}\ \mathit{owner} \\
\mathit{field} & ::= & t\ fd \\
\mathit{meth} & ::= & t\ mn\langle \mathit{formal}^*\rangle((t\ p)^*)\ \mathsf{where}\ \mathit{constr}^*\ \{\ e\ \} \\
t & ::= & c \mid \mathsf{int} \mid \mathsf{RHandle}\langle r\rangle \\
\mathit{formal} & ::= & k\ fn \\
k & ::= & \mathsf{Owner} \mid \mathsf{ObjOwner} \mid \mathit{rkind} \\
\mathit{rkind} & ::= & \mathsf{Region} \mid \mathsf{GCRegion} \mid \mathsf{NoGCRegion} \mid \mathsf{LocalRegion} \\
e & ::= & \mathsf{new}\ c \mid v \mid \mathsf{let}\ v = e\ \mathsf{in}\ \{\ e\ \} \mid v.fd \mid v.fd = v \mid v.mn\langle \mathit{owner}^*\rangle(v^*) \mid (\mathsf{RHandle}\langle r\rangle\ h)\ \{\ e\ \} \\
h & ::= & v \\
 & & \\
cn & \in & \text{class names} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
fn & \in & \text{formal identifiers} \\
v, p & \in & \text{variable names} \\
r & \in & \text{region identifiers}
\end{array}
$$

Figure 6-4: Grammar to Support Regions in Object-Oriented Programs

Grammar

To simplify the presentation of key ideas, we describe our type system as an extension to the core SafeJava type system we presented in Section 2.2 in Chapter 2. A SafeJava program is a sequence of class definitions followed by an initial expression. A predefined class Object is the root of the class hierarchy. Figure 6-4 presents grammar to support regions.

Owner Polymorphism

Every class definition is parameterized with one or more owners. An owner can be an object or a region. Parameterization allows programmers to implement a generic class whose objects can have different owners. The first formal owner is special: it owns the corresponding object; the other owners propagate the ownership information. Methods can also declare an additional list of formal owner parameters. Each time new formals are introduced, programmers can specify constraints between them using where clauses [50]. The constraints have the form "o_1 owns o_2" (i.e., $o_1 \succeq o_2$) and "o_1 outlives o_2" (i.e., $o_1 \succeq_r o_2$).

Each formal has an owner kind. There is a subkinding relation between owner kinds, resulting in the kind hierarchy from the upper half of Figure 6-5. The hierarchy is rooted in Owner, that has two subkinds: ObjOwner (owners that are objects; we avoid using Object because it is already used for the root of the class hierarchy) and Region. Region has two subkinds: GCRegion (the kind of the garbage collected heap) and NoGCRegion (the kind of other regions). Finally, NoGCRegion has a single subkind, LocalRegion. (At this point, there is no distinction between NoGCRegion and LocalRegion. We will add new kinds in the next section.)

Region Creation

The expression "(RHandle$\langle r\rangle$ h) $\{e\}$" creates a new region and introduces two identifiers r and h that are visible inside the scope of e. r is an owner of kind LocalRegion that is bound to the newly created region. h is a runtime value of type RHandle$\langle r\rangle$ that is

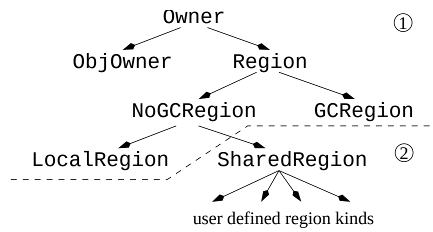

Figure 6-5: Owner Kind Hierarchy

bound to the *handle* of the region r. The region name r is only a compile-time entity; it is erased (together with all the ownership and region type annotations) immediately after typechecking. However, the region handle h is required at runtime when we allocate objects in region r (object allocation is explained in the next paragraph).

The newly created region is outlived by all regions that existed when it was created; it is destroyed at the end of the scope of e. This implies a "last in first out" order on region lifetimes. As we mentioned before, in addition to the user created regions, we have special regions: the garbage collected region heap (with handle $\mathsf{h}_{\mathsf{heap}}$) and the "immortal" region immortal (with handle $\mathsf{h}_{\mathsf{immortal}}$). Objects allocated in the immortal region are never deallocated. heap and immortal are never destroyed; hence, they outlive all regions.

We also allow methods to allocate objects in the special region initialRegion, which denotes the most recent region that was created before the method was called. We use runtime support to acquire the handle of initialRegion.

Object Creation

New objects are created using the expression "new $cn\langle o_{1..n}\rangle$". o_1 is the owner of the new object. (Recall that the first owner parameter always owns the corresponding object.) If o_1 is a region, the new object is allocated there; otherwise, it is allocated in the region where the object o_1 is allocated. For the purpose of typechecking, region handles are unnecessary. However, at runtime, we need the handle of the region we allocate in. The typechecker checks that we can obtain such a handle (more details are in Section 6.4). If o_1 is a region r, the handle of r must be in the environment. Therefore, if a method has to allocate memory in a specific region that is passed to it as an owner parameter, then it also needs to receive the corresponding region handle as an argument.

A formal owner parameter can be instantiated with an in-scope formal, a region name, or the this object. For every type $cn\langle o_{1..n}\rangle$ with multiple owners, SafeJava statically enforces the constraint that $o_i \succeq_r o_1$, for all $i \in \{1..n\}$. In addition, if an object of type $cn\langle o_{1..n}\rangle$ has a method mn, and if a formal owner parameter of mn is instantiated with an object obj, then SafeJava ensures that $obj \succeq_r o_1$. These constraints are necessary to statically enforce object encapsulation (as illustrated in Figure 2-14 in Chapter 2) and prevent dangling references (as illustrated at the end of Section 6.1) in the presence of subtyping.

Example

We illustrate our type system with the example in Figure 6-6. A TStack is a stack of T objects. It is implemented using a linked list. The TStack class is parameterized by

```
class TStack<Owner stackOwner, Owner TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>;
    newNode.init(value, head);  head = newNode;
  }

  T<TOwner> pop() {
    if(head == null) return null;
    T<TOwner> value = head.value;  head = head.next;
    return value;
  }
}

class TNode<Owner nodeOwner, Owner TOwner> {
  T<TOwner> value;
  TNode<nodeOwner, TOwner> next;

  void init(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next = n;
  }
}

(RHandle<r1> h1) {
  (RHandle<r2> h2) {
    TStack<r2,       r2>       s1;
    TStack<r2,       r1>       s2;
    TStack<r1,       immortal> s3;
    TStack<heap,     immortal> s4;
    TStack<immortal, heap>     s5;
 /* TStack<r1,       r2>       s6; illegal! */
 /* TStack<heap,     r1>       s7; illegal! */
}}
```

Figure 6-6: Stack of T Objects

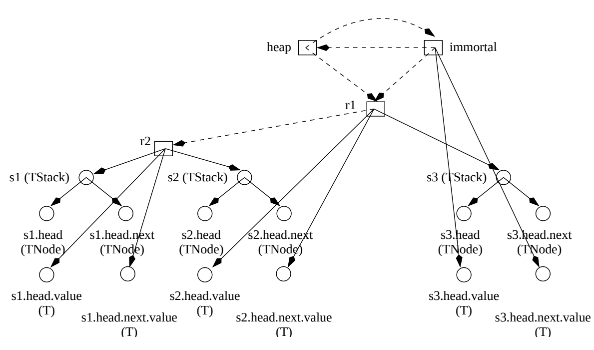

Figure 6-7: Ownership and Outlives Relations for TStacks s1, s2, s3

stackOwner and TOwner. stackOwner owns the TStack object and TOwner owns the T objects contained in the TStack. The code specifies that the TStack object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the TStack object.

The program creates two regions r1 and r2 such that r1 outlives r2. The program declares several TStack variables: the type of TStack s1 specifies that it is allocated in region r2 and so are the T objects in s1; TStack s2 is allocated in region r2 but the T objects in s2 are allocated in region r1; etc. Note that the type of s6 is illegal. This is because s6 is declared as TStack⟨r1,r2⟩, and r2 $\not\succeq_r$ r1. (Recall that in any legal type $cn\langle o_{1..n}\rangle$ with multiple owners, $o_i \succeq_r o_1$ for all $i \in \{1..n\}$.)

Figure 6-7 presents the ownership and the outlives relations from this example (assuming the stacks contain two elements each). We use circles for objects, rectangles for regions, solid arrows for ownership, and dashed arrows for the outlives relation between regions.

Safety Guarantees

The following two theorems state our safety guarantees. Part 1 of Theorems 5 and 6 state the object encapsulation property. Note that objects owned by regions are not encapsulated within other objects. Part 2 of Theorem 5 states the memory safety property.

THEOREM 5. *If objects o_1 and o_2 are allocated in regions r_1 and r_2 respectively, and field fd of o_1 points to o_2, then*

1. *Either owner of $o_2 \succeq o_1$, or owner of o_2 is a region.*
2. *Region r_2 outlives region r_1.*

PROOF. Suppose *class* $cn\langle f_{1..n}\rangle\{...\ T\langle x_1, ...\rangle\ fd\ ...\}$ is the class of o_1. Field fd of type $T\langle x_1, ...\rangle$ contains a reference to o_2. x_1 must therefore own o_2. x_1 can be either 1) heap, or 2) immortal, or 3) this, or 4) f_i, a class formal. In the first two cases, (owner of o_2) $= x_1$ is a region, and $r_2 = x_1 \succeq_r r_1$. In Case 3, (owner of o_2) $= o_1 \succeq o_1$, and $r_2 = r_1 \succeq_r r_1$. In Case 4, we know that $f_i \succeq_r f_1$, since all owners in a legal type outlive the first owner. Therefore, (owner of o_2) $= x_1 = f_i \succeq_r f_1 \succeq_r$ this $= o_1$. If (owner of o_2) is an object, we know from Lemma 3 that (owner of o_2) $\succeq o_1$. This also implies that $r_2 = r_1 \succeq_r r_1$. If the (owner of o_2) is a region, we know from Lemma 4 that there exists region r such that (owner of o_2) $\succeq_r r$ and $r \succeq o_1$. Therefore $r_2 = r \succeq_r r_1$. □

THEOREM 6. *If a variable v in a method mn of an object o_1 points to an object o_2, then*

1. *Either owner of $o_2 \succeq o_1$, or owner of o_2 is a region.*

PROOF. Similar to the proof of Theorem 5, except that (owner of o_2) can now also be: 5) a region declared within the method, or 6) a formal method parameter that is a region or initialRegion (that is not required to outlive o_1), or 7) a formal method parameter that is an object. In Cases 5 and 6, (owner of o_2) is a region. In Case 7, (owner of o_2) $\succeq o_1$. The first four cases are identical. □

```
class Foo<q> { int x = 0;  void accessMe() { x++; } }

class SuperType<r> { void m() {} }

class SubType<r,q> extends SuperType<r> {
  Foo<q>             region_q_contains_me;
  SubType(Foo<q> x) {region_q_contains_me = x;}
  void m()          {region_q_contains_me.accessMe();}
}


(RHandle<r> hr) {
  SuperType<r> s;

  (RHandle<q> hq) {
    Foo<q> f = new Foo<q>();
    s = new SubType<r,q>(f);  // SubType<r,q> is an illegal type in SafeJava
  }                           // because q does not outlive r

  s.m();                      // Violates memory safety
}
```

Figure 6-8: Violation of Memory Safety in an Unsound Type System

Most previous region type systems allow programs to create, but not follow, dangling references. Such references can cause a safety problem when used with copying collectors. SafeJava therefore prevents a program from creating dangling references in the first place. Part 2 of Theorem 5 prevents object fields from containing dangling references. Even though Theorem 6 does not have a similar Part 2, we can prove, using lexical scoping of region names, that local variables cannot contain dangling references either.

Note that to have a sound type system that statically prevents dangling references, it is necessary to have the constraint that in any legal type $cn\langle o_{1..n}\rangle$ with multiple owners, $o_i \succeq_r o_1$ for all $i \in \{1..n\}$. Figure 6-8 illustrates this point with an example in a type system without this constraint. In the figure, the program creates a Foo object in region q, stores a pointer to it in object s, and deletes the region q, thus creating a dangling reference from s to the Foo object. The program then invokes a method s.m which tries to access the Foo object, thus violating memory safety.

6.2 Regions for Multithreaded Programs

This section describes how we support multithreaded programs. Figure 6-9 presents the language extensions. A fork instruction spawns a new thread that evaluates the invoked method. The evaluation is performed only for its effect; the parent thread does not wait for the completion of the new thread and does not use the result of the method call. Our unstructured concurrency model (similar to Java's model) is incompatible with the regions from Section 6.1 whose lifetimes are lexically bound. Those regions can still be used for allocating thread-local objects (hence the name of the associated region kind, LocalRegion), but objects shared by multiple threads require *shared* regions, of kind SharedRegion.

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^* \ \mathit{srkdef}^* \ e \\
\mathit{srkdef} & ::= & \textsf{regionKind} \ \mathit{srkn}\langle \mathit{formal}^* \rangle \ \textsf{extends} \ \mathit{srkind} \, \textsf{where} \ \mathit{constr}^* \ \{ \ \mathit{field}^* \ \mathit{subsreg}^* \ \} \\
\mathit{rkind} & ::= & \text{... as in Figure 6-4 ...} \mid \mathit{srkind} \\
\mathit{srkind} & ::= & \mathit{srkn}\langle \mathit{owner}^* \rangle \mid \textsf{SharedRegion} \\
\mathit{subsreg} & ::= & \mathit{srkind} \ \mathit{rsub} \\
 & & \\
e & ::= & \text{... as in Figure 6-4 ...} \mid \\
 & & \textsf{fork} \ v.mn\langle \mathit{owner}^* \rangle (v^*) \mid (\textsf{RHandle}\langle \mathit{rkind} \ r \rangle \ h) \ \{ \ e \ \} \mid (\textsf{RHandle}\langle r \rangle \ h = [\textsf{new}]_{\mathrm{opt}} \ h.\mathit{rsub}) \ \{ \ e \ \} \mid \\
 & & h.\mathit{fd} \mid h.\mathit{fd} = v \\
 & & \\
\mathit{srkn} & \in & \text{shared region kind names} \\
\mathit{rsub} & \in & \text{shared subregion names}
\end{array}
$$

Figure 6-9: Grammar Extensions to Support Regions in Multithreaded Programs

Shared Regions

"(RHandle$\langle rkind\ r\rangle\ h$) $\{e\}$" creates a shared region ($rkind$ specifies the region kind of r; region kinds are explained later in this section). Inside expression e, the identifiers r and h are bound to the region and the region handle, respectively. Inside e, r and h can be passed to child threads. The objects allocated inside a shared region are not deleted as long as some thread can still access them. To ensure this, each thread maintains a stack of shared regions it can access, and each shared region maintains a counter of how many such stacks it is an element of. When a new shared region is created, it is pushed onto the region stack of the current thread and its counter is initialized to one. A child thread inherits all the shared regions of its parent thread; the counters of these regions are incremented when the child thread is forked. When the scope of a region name ends (the names of the shared regions are still lexically scoped, even if the lifetimes of the regions are not), the corresponding region is popped off the stack and its counter is decremented. When a thread terminates, the counters of all the regions from its stack are decremented. When the counter of a region becomes zero, the region is deleted. The typing rule for a fork expression checks that objects allocated in local regions are not passed to the child thread as arguments; it also checks that local regions and handles to local regions are not passed to the child thread.

Subregions and Portals

Shared regions provide the basis for inter-thread communication. However, in many cases, they are not enough. E.g., consider two long-lived threads, a producer and a consumer, that communicate through a shared region in a repetitive way. In each iteration, the producer allocates some objects in the shared region and the consumer subsequently uses the objects. These objects become unreachable after each iteration. However, these objects are not deleted until both threads terminate and exit the shared region. To prevent this memory leak, we allow shared regions to have subregions. In each iteration, the producer and the consumer can enter a subregion of the shared region and use it for communication. At the end of the iteration, both the threads exit the subregion and the reference count of the subregion goes to zero—the objects in the subregion are thus deleted after each iteration.

We must also allow the producer to pass references to objects it allocates in the subregion in each iteration to the consumer. Note that storing the references in the fields of a "hook" object is not possible: objects allocated outside the subregion cannot point to objects in the subregion (otherwise, those references would result in dangling references when objects in the subregion are deleted), and objects allocated in the subregion do not survive between iterations and hence cannot be used as "hooks". To solve this problem, we allow (sub)regions to contain *portal* fields. A thread can store the reference to an object in a portal field; other threads can then read the portal field to obtain the reference.

Region Kinds

In practice, programs can declare several shared region kinds. Each kind extends another shared region kind and can declare several portal fields and subregions (see grammar rule for *srkdef* in Figure 6-9). The resulting shared region kind hierarchy has SharedRegion as its root. The owner kind hierarchy now includes both Areas 1 and 2 from Figure 6-5. Similar to classes, shared region kinds can be parameterized with owners; however, unlike objects, regions do not have owners so there is no special meaning attached to the first owner.

Expression "$(\mathsf{RHandle}\langle r_2\rangle\ h_2 = [\mathsf{new}]_{\text{opt}}\ h_1.rsub)\ \{e\}$" evaluates e in an environment where r_2 is bound to the subregion $rsub$ of the region r_1 that h_1 is the handle of, and h_2 is bound to the handle of r_2. In addition, if the keyword new is present, r_2 is a newly created subregion, distinct from the previous $rsub$ subregion.

If h is the handle of region r, the expression "$h.fd$" reads r's portal field fd, and "$h.fd = v$" stores a value into that field. The rule for portal fields is the same as that for object fields: a portal field of a region r is either null or points to an object allocated in r or in a region that outlives r.

Flushing Subregions

When all the objects in a subregion become inaccessible, the subregion is flushed, i.e., all objects allocated inside it are deleted. We do not flush a subregion if its counter is positive. Furthermore, we do not flush a subregion r if any of its portal fields is non-null (to allow some thread to enter it later and use those objects) or if any of r's subregions has not been flushed yet (because the objects in those subregions might point to objects in r). Recall that subregions are a way of "packaging" some data and sending it to another thread; the receiver thread looks inside the subregion (starting from the portal fields) and uses the data. Therefore, as long as a subregion with non-null portal fields is reachable (i.e., a thread may obtain its handle), the objects allocated inside it can be reachable even if no thread is currently in the subregion.

Example

Figure 6-10 contains an example that illustrates the use of subregions and portal fields. The main thread creates a shared region of kind BufferRegion and then starts two threads, a producer and a consumer, that communicate through the shared region. In each iteration, the producer enters subregion b (of kind BufferSubRegion), allocates a Frame object in it, and stores a reference to the frame in subregion's portal field f. Next, the producer exits the subregion and waits for the consumer. The subregion is not flushed because the portal field f is non-null. The consumer then enters the subregion, uses the frame object pointed to by its portal field f, sets f to null, and exits the subregion. Now, the subregion is flushed (because its counter is zero and all its fields are null) and a new iteration starts. (We do not discuss synchronization issues here in this Chapter.)

```
regionKind BufferRegion extends SharedRegion {
  BufferSubRegion b;
}

regionKind BufferSubRegion extends SharedRegion {
  Frame<this> f;
}

class Producer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = new Frame<r2>;
        get_image(frame);
        h2.f = frame;
      }
      ... // wake up the consumer
      ... // wait for the consumer
}}}

class Consumer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      ... // wait for the producer
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = h2.f;
        h2.f = null;
        process_image(frame);
      }
      ... // wake up the producer
}}}

(RHandle<BufferRegion r> h) {
  fork (new Producer<r>).run(h);
  fork (new Consumer<r>).run(h);
}
```

Figure 6-10: Producer Consumer Example

6.3 Regions for Real-Time Programs

A real-time program consists of a set of real-time threads, a set of regular threads, and a special garbage collector thread. (This is a conceptual model; actual implementations might differ.) A real-time thread has strict deadlines for completing its tasks.[1]

Figure 6-11 presents the language extensions to support real-time programs. The expression "RT_fork $v.mn\langle owner^*\rangle(v^*)$" spawns a new real-time thread to evaluate mn. Such a thread cannot afford to be interrupted for an unbounded amount of time by the garbage collector—the rest of this section explains how SafeJava statically ensures this property.

[1]Our terminology is related, but not identical to the RTSJ terminology. E.g., our real-time threads are similar to (and more restrictive than) the RTSJ NoHeapRealtimeThreads.

$$
\begin{array}{rcl}
\mathit{meth} & ::= & t\ mn\langle \mathit{formal}^*\rangle((t\ p)^*)\ \mathit{effects}\ \textsf{where}\ \mathit{constr}^*\ \{e\} \\
\mathit{effects} & ::= & \textsf{accesses}\ \mathit{owner}^* \\
\mathit{owner} & ::= & \text{... as in Figure 6-4 ...} \mid \textsf{RT} \\
\mathit{subsreg} & ::= & \mathit{srkind} : \mathit{rpol}\ \mathit{tt}\ \mathit{rsub} \\
\mathit{rpol} & ::= & \textsf{LT}(\mathit{size}) \mid \textsf{VT} \\
\mathit{tt} & ::= & \textsf{RT} \mid \textsf{NoRT} \\
k & ::= & \text{... as in Figure 6-4 ...} \mid \mathit{rkind} : \textsf{LT} \\
 & & \\
e & ::= & \text{... as in Figure 6-9 ...} \mid \\
 & & (\textsf{RHandle}\langle \mathit{rkind} : \mathit{rpol}\ r\rangle\ h)\ \{\ e\ \} \mid \textsf{RT\_fork}\ v.mn\langle \mathit{owner}^*\rangle(v^*)
\end{array}
$$

Figure 6-11: Grammar Extensions to Support Regions in Realtime Programs

Effects

The garbage collector thread must synchronize with any thread that creates or destroys heap roots, i.e., references to heap objects, otherwise it might end up collecting reachable objects. Therefore, we must ensure that the real-time threads do not read or overwrite references to heap objects. (The last restriction is needed to support copying collectors.) To statically check this, we allow methods to declare *effects* clauses [106]. In our system, the effects clause of a method lists the owners (some of them regions) that the method *accesses*. Accessing a region means allocating an object in that region. Accessing an object means reading or overwriting a reference to that object or allocating another object owned by that object. Note that we do not consider reading or writing a field of an object as accessing that object. If a method's effects clause consists of owners $o_{1..n}$, then any object or region accessed by that method, the methods it invokes, and the threads it spawns (transitively) is guaranteed to be outlived by o_i, for some $i \in \{1..n\}$.

The typing rule for an RT_fork expression checks all the constraints of a regular fork expression. In addition, it checks that references to heap objects are not passed as arguments to the new thread, and that the effects clause of the method evaluated in the new thread does not contain the heap region or any object allocated in the heap region. If an RT_fork expression typechecks, the new real-time thread cannot receive any heap reference. Furthermore, it cannot create a heap object, or read or overwrite a heap reference in an object field—the type system ensures that in each of the above cases, the heap region or an object allocated in the heap region appears in the method effects.

Region Allocation Policies

A real-time thread cannot create an object if this operation requires allocating new memory, because allocating memory requires synchronization with the garbage collector. A real-time thread can, however, create an object in a preallocated memory region.

SafeJava supports two allocation policies for regions. One policy is to allocate memory on demand (potentially in large chunks), as new objects are created in the region. Allocating a new object can take unbounded time or might not even succeed (if a new chunk is needed and the system runs out of memory). Flushing the region frees all the memory allocated for that region. Following the RTSJ terminology, we call such regions *VT* (Variable Time) regions.

The other policy is to allocate all the memory for a region at region creation time. The programmer must provide an upper bound for the total size of the objects that will be allocated in the region. Allocating an object requires sliding a pointer—if the region is already full, the system throws an exception to signal that the region size was too small. Allocating a new object takes time linear in its size: sliding the pointer takes constant time, but we also have to set to zero each allocated byte. Flushing the region simply resets a pointer, and, importantly, *does not* free the memory allocated for the region. We call regions that use this allocation policy *LT* (Linear Time) regions. Once we have an LT subregion, threads can repeatedly enter it, allocate objects in it, exit it (thus flushing it), and re-enter it without having to allocate new memory. This is possible because flushing an LT region does not free its memory. LT subregions are thus ideal for real-time threads: once such a subregion is created (with a large enough size), all object creations will succeed, in linear time; moreover, the subregion can be flushed and reused without memory allocation.

We allow users to specify the region allocation policy (LT or VT) when a new region is created. The policy for subregions is declared in the shared region kind declarations. When a user specifies an LT policy, the user also has to specify the size of the region (in bytes). An expression "$(\mathsf{RHandle}\langle rkind : rpol\ \ r\rangle\ h)\ \{e\}$" creates a region with allocation policy *rpol* and allocates memory for all its (transitive) LT (sub)regions (including itself). SafeJava checks that a region has a finite number of transitive subregions.

If a method enters a VT region or a top level region (i.e., a region that is not a subregion), the typechecker ensures that the method contains the heap region in its effects clause. This is to prevent real-time threads from invoking such methods. However, a method that does not contain the heap region in its effects clause can still enter an existing LT subregion, because no memory is allocated in that case.

Preventing the RTSJ Priority Inversion

So far, we presented techniques for checking that real-time threads do not create or destroy heap references, create new regions, or allocate objects in VT regions. However, there are two other subtle ways a thread can interact with the garbage collector.

First, the garbage collector needs to know all locations that refer to heap objects, including locations that are inside regions. Suppose a real-time thread uses an LT region that contains such heap references (created by a non-real-time thread). The real-time thread can flush the region (by exiting it) thus destroying any heap reference that existed in the region. If we use a copying garbage collector, the real-time thread has to interact with the garbage collector to inform it about the destruction of those heap references. Therefore, we should prevent regions that can be flushed by a real-time thread from containing any heap reference (even if the reference is not explicitly read or overwritten by the real-time thread). Note that this restriction is relevant only for subregions: a real-time thread cannot create a top-level region and hence cannot flush a top-level region either.

Second, when a thread enters or exits a subregion, it needs to do some bookkeeping. To preserve the integrity of the runtime region implementation, some synchronization is necessary during this bookkeeping. E.g., when a thread exits a subregion, the test that the subregion can be flushed and the actual flushing have to be executed atomically, without allowing any thread to enter the subregion "in between". If a regular thread exiting a subregion is suspended by the garbage collector, then a real-time thread entering the subregion might have to wait for an unbounded amount of time.

The above *priority inversion* problem occurs even in the RTSJ. To prevent these subtle interactions, we impose the restriction that real-time threads and regular threads cannot share subregions. Subregions used by real-time threads thus cannot contain heap references, and real-time threads never have to wait for unbounded amounts of time.

For each subregion, programmers specify in the region kind definitions whether the subregion will be used only by real-time threads (RT subregions) or only by regular threads (NoRT subregions). Note that real-time and regular threads can still communicate using top-level regions. Any method that enters an RT subregion must contain the special effect RT in its effects clause. Any method that enters a NoRT subregion must contain the heap region in its effects clause. The type system checks that no regular thread can invoke a method that has an RT effect, and no real-time thread can invoke a method that has a heap effect.

6.4 Formal Description

Previous sections presented the grammar for our core language in Figures 6-4, 6-9, and 6-11. This section presents some sample typing rules. Appendix 6.A at the end of this chapter contains all the rules.

The core of our type system is a set of typing judgments of the form $P; E; X; r_{cr} \vdash e : t$. P, the program being checked, is included to provide information about class definitions. The typing environment E provides information about the type of the free variables of e ($t\ v$, i.e., variable v has type t), the kind of the owners currently in scope ($k\ o$, i.e., owner o has kind k), and the two relations between owners: the "ownership" relation ($o_2 \succeq o_1$, i.e., o_2 owns o_1) and the "outlives" relation ($o_2 \succeq_r o_1$, i.e., o_2 outlives o_1). More formally, $E ::= \emptyset \mid E, t\ v \mid E, k\ o \mid E, o_2 \succeq o_1 \mid E, o_2 \succeq_r o_1$. r_{cr} is the current region. X must subsume the effects of e. t is the type of the expression e.

A useful auxiliary rule is $E \vdash X_1 \succeq_r X_2$, i.e., the effects X_1 *subsume* the effects X_2: $\forall o \in X_2$, $\exists g \in X_1$, s.t. $g \succeq_r o$. To prove constraints of the form $g \succeq_r o$, $g \succeq o$ etc. in a specific environment E, the checker uses the constraints from E, and the properties of $\succeq_r$ and $\succeq$: transitivity, reflexivity, $\succeq$ implies $\succeq_r$, and the fact that the first owner from the type of an object owns the object.

The expression "$(\mathsf{RHandle}\langle r\rangle\ h)\ \{e\}$" creates a local region and evaluates e in an environment where r and h are bound to the new region and its handle respectively. The associated typing rule is presented below:

[EXPR LOCAL REGION]

$$\frac{\begin{array}{c} E_2 = E,\ \mathsf{LocalRegion}\ r,\ \mathsf{RHandle}\langle r\rangle\ h,\ (r_e \succeq_r r)_{\forall r_e \in Regions(E)} \\ P \vdash_{\text{env}} E_2 \qquad P; E_2; X, r; r \vdash e : t \qquad E \vdash X \succeq_r \mathsf{heap} \end{array}}{P; E; X; r_{cr} \vdash (\mathsf{RHandle}\langle r\rangle\ h)\ \{e\} : \mathsf{int}}$$

The rule starts by constructing an environment E_2 that extends the original environment E by recording that r has kind LocalRegion and h has type $\mathsf{RHandle}\langle r\rangle$. As r is deleted at the end of e, all existing regions outlive it; E_2 records this too ($Regions(E)$ denotes the set of all regions from E). e should typecheck in the context of the environment E_2 and the permitted effects are X, r (the local region r is a permitted effect inside e). Because creating a region requires memory allocation, X must subsume heap. The expression is evaluated only for its side-effects and its result is never used. Hence, the type of the entire expression is int.

The rule for a field read expression "$v.fd$" first finds the type $cn\langle o_{1..n}\rangle$ for v. Next, it verifies that fd is a field of class cn; let t be its declared type. The rule obtains the type of the entire expression by substituting in t each formal owner parameter fn_i of cn with the corresponding owner o_i:

[EXPR REF READ]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \\ ((t' = \mathsf{int})\ \vee\ (t' = cn'\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq_r o'_1)) \end{array}}{P; E; X; r_{cr} \vdash v.fd : t'}$$

The last line of the rule checks that if the expression reads an object reference (i.e., not an integer), then the list of effects X subsumes the owner of the referenced object.

For an object allocation expression "$\mathsf{new}\ cn\langle o_{1..n}\rangle$", the rule first checks that class cn is defined in P:

[EXPR NEW]

$$\frac{\begin{array}{c} \mathsf{class}\ cn\langle (k_i\ fn_i)_{i\in\{1..n\}}\rangle\ ...\ \mathsf{where}\ constr_{1..c}\ ... \in P \\ \forall i = 1..m,\ (E \vdash_{\mathrm{k}} o_i : k'_i\ \wedge\ P \vdash k'_i \leq_{\mathrm{k}} k_i\ \wedge\ E \vdash o_i \succeq_r o_1) \\ \forall i = 1..c,\ E \vdash constr_i[o_1/fn_1]..[o_m/fn_m] \\ E \vdash X \succeq_r o_1 \qquad E \vdash_{\mathrm{av}} \mathsf{RH}(o_1) \end{array}}{P; E; X; r_{cr} \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

Next, it checks that each formal owner parameter fn_i of cn is instantiated with an owner o_i of appropriate kind, i.e., the kind k'_i of o_i is a subkind of the declared kind k_i of fn_i. It also checks that in E, each owner o_i outlives the first owner o_1, and each constraint of cn is satisfied. Allocating an object means accessing its owner; therefore, X must subsume o_1. The new object is allocated in the region o_1 (if o_1 is a region) or in the region that o_1 is allocated in (if o_1 is an object). The last part of the precondition, $E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)$, checks that the handle for this region is available. To prove facts of this kind, the type system uses the following rules:

[AV HANDLE]

$$\frac{E = E_1,\ \mathsf{RHandle}\langle r\rangle\ h,\ E_2}{E \vdash_{\mathrm{av}} \mathsf{RH}(r)}$$

[AV THIS]

$$\frac{}{E \vdash_{\mathrm{av}} \mathsf{RH}(\mathsf{this})}$$

[AV TRANS1]

$$\frac{E \vdash o_1 \succeq o_2 \qquad E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}$$

[AV TRANS2]

$$\frac{E \vdash o_1 \succeq o_2 \qquad E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}$$

The rule [AV HANDLE] looks for a region handle in the environment. The environment always contains handles for heap and $\mathsf{immortal}$; in addition, it contains all handle identifiers that are in scope. The rule [AV THIS] reflects the fact that our runtime is able to find the handle of the region where an object (this in particular) is allocated. The last two rules use the fact that all objects are allocated in the same region as their owner. Therefore, if $o_1 \succeq o_2$ and the region handle for one of them is available, then the region handle for the other one is also available. Note that these rules do significant reasoning, thus reducing annotation burden; e.g., if a method allocates only objects (transitively) owned by this, it does not need an explicit region handle argument.

We end this section with the typing rule for `fork`. The rule first checks that the method call is well-typed. (see rule [EXPR INVOKE] in Appendix 6.A). Note that mn cannot have the RT effect: a non-real-time thread cannot enter a subregion that is reserved only for real-time threads.

[EXPR FORK]

$$\frac{\begin{array}{c} P; E; X \setminus \{\mathsf{RT}\}; r_{cr} \vdash v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \\ \mathit{NonLocal}(k) \stackrel{def}{=} (P \vdash k \leq_{\mathrm{k}} \mathsf{SharedRegion}) \ \vee\ (P \vdash k \leq_{\mathrm{k}} \mathsf{GCRegion}) \\ E \vdash \mathrm{RKind}(r_{cr}) = k_{cr} \qquad \mathit{NonLocal}(k_{cr}) \\ P; E; X; r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i = 1..m,\ (E \vdash \mathrm{RKind}(o_i) = k_i \ \wedge\ \mathit{NonLocal}(k_i)) \end{array}}{P; E; X; r_{cr} \vdash \mathsf{fork}\ v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \mathsf{int}}$$

The rule checks that the new thread does not receive any local region or objects allocated in a local region. It uses the following observation: the only owners that appear in the types of the method arguments are: `initialRegion`, this, the formals for the method and the formals for the class the method belongs to. Therefore, the arguments passed to the method mn from the fork instruction may be owned only by the current region at the point of the fork, by the owners $o_{1..n}$ that appear in the type of the object v_0 points to, or by the owners $o_{(n+1)..m}$ that appear explicitly in the fork instruction. For each such owner o, our system uses the rule $E \vdash \mathrm{RKind}(o) = k$ to extract the kind k of the region it stands for (if it is a region), or of the region it is allocated in (if it is an object). The rule next checks that k is a subkind of SharedRegion or GCRegion. The rules for inferring statements of the form $E \vdash \mathrm{RKind}(o_i) = k_i$ (see Appendix 6.A) are similar to the previously explained rules for checking that a region handle is available. The key idea they exploit is that a subobject is allocated in the same region as its owner.

6.5 Type Inference

Although SafeJava is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information that SafeJava requires. Instead, we use a combination of type inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. SafeJava also supports user-defined defaults to cover specific patterns that might occur in user code. We emphasize that our approach to inference is purely intraprocedural and we do not infer method signatures or types of instance variables. Rather, we use a default completion of partial type specifications in those cases. This approach permits separate compilation.

The following are some defaults currently provided by SafeJava. If owners of method local variables are not specified, we use a simple unification-based approach to infer the owners. We described this approach in Section 4.5 in Chapter 4. For parameters unconstrained after unification, we use initialRegion. For unspecified owners in method signatures, we use initialRegion as the default. For unspecified owners in instance variables, we use the owner of this as the default. For static fields, we use immortal as the default. Our default accesses clauses contain all class and method owner parameters and initialRegion.

6.6 Translation to Real-Time Java

Although our system provides significant improvements over the RTSJ, programs in our language can be translated to RTSJ reasonably easily, by local translation rules. This is mainly because we designed our system so that it can be implemented using type erasure (region handles exist specifically for this purpose). Also, RTSJ has mechanisms that are

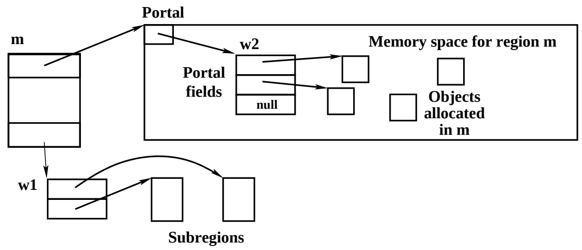

Figure 6-12: Translation of a Region With Three Fields and Two Subregions.

powerful enough to support our features. RTSJ offers LTMemory and VTMemory regions where it takes linear time and variable time (respectively) to allocate objects. RTSJ regions are Java objects that point to some memory space. In addition, RTSJ has two special regions: heap and immortal. A thread can allocate in the current region using new. A thread can also allocate in any region that it entered using newInstance, which requires the corresponding region object. RTSJ regions are maintained similarly to our shared regions, by counting the number of threads executing in them. RTSJ regions have one *portal*, which is similar to a portal field except that its declared type is Object. Most of the translation effort is focused on providing the missing features: subregions and multiple, typed portal fields. We discuss the translation of several important features from our type system; the full translation is discussed in Appendix 6.B at the end of this chapter.

We represent a region r from our system as an RTSJ region m plus two auxiliary objects w1 and w2 (see Figure 6-12). m points to a memory area that is pre-allocated for an LT region, or grown on-demand for a VT region. m also points to an object w1 whose fields point to the representation of r's subregions. (We subclass LT/VTMemory to add an extra field.) In addition, m's portal points to an object w2 that serves as a wrapper for r's portal fields. w2 is allocated in the memory space attached to m, while m and w1 are allocated in the region that was current at the time m was created.

The translation of "new $cn\langle o_{1..n}\rangle$" requires a reference to (i.e., the handle of) the region we allocate in. If this is the same as the current region, we use the more efficient new. The type rules already proved that we can obtain the necessary handle, i.e., $E \vdash_{\text{av}} \mathsf{RH}(o_1)$; we presented the relevant type rules in Section 6.4. Those rules "pushed" the judgment $E \vdash_{\text{av}} \mathsf{RH}(o)$ up and down the ownership relation until we obtained an owner whose region handle was available: immortal, heap, this, or a region whose region handle was available in a local variable. RTSJ provides mechanisms for retrieving the handle in the first three cases: ImmortalArea.instance(), HeapArea.instance(), and MethodArea.getMethodArea(Object), respectively. In the last case, we simply use the handle from the local variable.

6.7 Programming Experience

To gain preliminary experience, we implemented several programs in SafeJava. These include two micro benchmarks (Array and Tree), two scientific computations (Water and Barnes), several components of an image recognition pipeline (load, cross, threshold, hysteresis, and thinning), and several simple servers (http, game, and phone, a database-backed information sever). In our implementations, the primary data structures are allocated in regions (i.e., not in the garbage collected heap). In each case, once we understood how

Program	Lines of Code	Lines Changed
Array	56	4
Tree	83	8
Water	1850	31
Barnes	1850	16
ImageRec	567	8
http	603	20
game	97	10
phone	244	24

Figure 6-13: Programming Overhead

Program	Execution Time (sec)		Overhead
	Static Checks	Dynamic Checks	
Array	2.24	16.2	7.23
Tree	4.78	23.1	4.83
Water	2.06	2.55	1.24
Barnes	19.1	21.6	1.13
ImageRec	6.70	8.10	1.21
load	0.667	0.831	1.25
cross	0.014	0.014	1.0
threshold	0.001	0.001	1
hysteresis	0.005	0.006	1
thinning	0.023	0.026	1.1
save	0.617	0.731	1.18

Figure 6-14: Dynamic Checking Overhead

the program worked and decided on the memory management policy to use, adding the extra type annotations was fairly straightforward. Figure 6-13 presents a measure of the programming overhead involved. It shows the number of lines of code that needed type annotations. In most cases, we only had to change code where regions were created.

We also used our RTSJ implementation to measure the execution times of these programs both with and without the dynamic checks specified in the Real-Time Specification for Java. Figure 6-14 presents the running times of the benchmarks both with and without dynamic checks. Note that there is no garbage collection overhead in any of these running times because the garbage collector never executes. Our micro benchmarks (Array and Tree) were written specifically to maximize the checking overhead—our development goal was to maximize the ratio of assignments to other computation. These programs exhibit the largest performance increases—they run approximately 7.2 and 4.8 times faster, respectively, without checks. The performance improvements for the scientific programs and image processing components provide a more realistic picture of the dynamic checking overhead. These programs have more modest performance improvements, running up to 1.25 times faster without the checks. For the servers, the running time is dominated by the network processing overhead and check removal has virtually no effect. We present the overhead of dynamic referencing and assignment checks in this thesis. For a detailed analysis of the performance of a full range of RTSJ features, see [46, 47].

6.8 Related Work

The seminal work in [129, 128] introduces a static type system for region-based memory management for ML. SafeJava extends this to object-oriented programs by combining the benefits of region types and ownership types in a unified type system framework. SafeJava extends region types to multithreaded programs by allowing long-lived threads to share objects without using the heap and without having memory leaks. SafeJava extends region types to real-time programs by ensuring that real-time threads do not interfere with the garbage collector.

One disadvantage with most region-based management systems is that they enforce a lexical nesting on region lifetimes; so objects allocated in a given region may become inaccessible long before the region is deleted. [4] presents an analysis that enables some regions to be deleted early, as soon as all of the objects in the region are unreachable. Other approaches include the use of linear types to control when regions are deleted [48, 52]. None of these approaches currently support object-oriented programs and the consequent subtyping, multithreaded programs with shared regions, or real-time programs with real-time threads (although it should be possible to extend them to do so). Conversely, it should also be possible to apply these techniques to SafeJava. In fact, existing systems already combine ownership-based type systems and unique pointers [44, 26, 6].

RegJava [39] has a region type system for object-oriented programs that supports subtyping and method overriding. Cyclone [80] is a dialect of C with a region type system. SafeJava improves on these two systems by combining the benefits of ownership types and region types in a unified framework. An extension to Cyclone handles multithreaded programs and provides shared regions [79]. SafeJava improves on this by providing subregions in shared regions and portal fields in subregions, so that long-lived threads can share objects without using the heap and without having memory leaks. Other systems for regions [72, 73] use runtime checks to ensure memory safety. These systems are more flexible, but they do not statically ensure safety.

To our knowledge, SafeJava is the first static type system for memory management in real-time programs. [54, 55] automatically translates Java code into RTSJ code using off-line dynamic analysis to determine the lifetime of an object. Unlike SafeJava, this system does not require type annotations. It does, however, impose a runtime overhead and it is not safe because the dynamic analysis might miss some execution paths. Programmers can use this analysis to obtain suggestions for region type annotations. [121] uses escape analysis to remove RTSJ runtime checks [122]. However, the analysis is effective only for programs in which no object escapes the computation that allocated it. SafeJava is more flexible: we allow a computation to allocate objects in regions that may outlive the computation.

Real-time garbage collection [12, 10] provides an alternative to region-based memory management for real-time programs. It has the advantage that programmers do not have to explicitly deal with memory management. The idea is to perform a fixed amount of garbage collection activity for a given amount of allocation. With fixed-size allocation blocks and in the absence of cycles, reference counting can deliver a real-time garbage collector that imposes no space overhead as compared with manual memory management. Copying and mark and sweep collectors, on the other hand, pay space to get bounded-time allocation. The amount of extra space depends on the maximum live heap size, the maximum allocation rate, and other memory management parameters. The additional space allows the collector to successfully perform allocations while it processes the heap to reclaim memory.

To obtain the real-time allocation guarantee, the programmer must calculate the required memory management parameters, then use those values to provide the collector with the required amount of extra space. In contrast, region-based memory management provides an explicit mechanism that programmers can use to structure code based on their understanding of the memory usage behavior of a program; this mechanism may enable programmers to obtain a smaller space overhead. The additional development burden consists of grouping objects into regions and determining the maximum size of LT regions [74, 75].

6.9 Conclusions

The Real-Time Specification for Java (RTSJ) allows programs to create real-time threads and use region-based memory management. The RTSJ uses runtime checks to ensure memory safety. This chapter describes how the SafeJava type system statically guarantees that these runtime checks will never fail for well-typed programs. SafeJava therefore 1) provides an important safety guarantee and 2) makes it possible to eliminate the runtime checks and their associated overhead.

SafeJava also makes several contributions over previous work on region types. For object-oriented programs, it combines the benefits of region types and ownership types in a unified type system framework. For multithreaded programs, it allows long-lived threads to share objects without using the heap and without having memory leaks. For real-time programs, it ensures that real-time threads do not interfere with the garbage collector.

Our experience indicates that SafeJava is sufficiently expressive and requires little programming overhead, and that eliminating the RTSJ runtime checks using a static type system can significantly decrease the execution time of real-time programs.

6.A Rules for Type Checking

This section formally describes our type system for region-based memory management. To simplify the presentation of the key ideas, it builds on the type system presented in Appendix 2.A at the end of Chapter 2. The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^* \ \mathit{srkdef}^* \ e \\
\mathit{defn} & ::= & \mathsf{class}\ \mathit{cn}\langle \mathit{formal}+\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ \mathit{constr}^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{formal} & ::= & k\ \mathit{fn} \\
c & ::= & \mathit{cn}\langle \mathit{owner}+\rangle \mid \mathsf{Object}\langle \mathit{owner}\rangle \\
\mathit{owner} & ::= & \mathit{fn} \mid r \mid \mathsf{this} \mid \mathsf{initialRegion} \mid \mathsf{heap} \mid \mathsf{immortal} \mid \mathsf{RT} \\
\mathit{field} & ::= & t\ \mathit{fd} \\
\mathit{meth} & ::= & t\ \mathit{mn}\langle \mathit{formal}^*\rangle((t\ p)^*)\ \mathit{effects}\ \mathsf{where}\ \mathit{constr}^*\ \{\ e\ \} \\
\mathit{effects} & ::= & \mathsf{accesses}\ \mathit{owner}^* \\
\mathit{constr} & ::= & \mathit{owner}\ \mathsf{owns}\ \mathit{owner} \mid \mathit{owner}\ \mathsf{outlives}\ \mathit{owner} \\
t & ::= & c \mid \mathsf{int} \mid \mathsf{RHandle}\langle r\rangle \\
\\
\mathit{srkdef} & ::= & \mathsf{regionKind}\ \mathit{srkn}\langle \mathit{formal}^*\rangle\ \mathsf{extends}\ \mathit{srkind}\ \mathsf{where}\ \mathit{constr}^*\ \{\ \mathit{field}^*\ \mathit{subsreg}^*\ \} \\
\mathit{subsreg} & ::= & \mathit{srkind} : \mathit{rpol}\ \ \mathit{tt}\ \ \mathit{rsub} \\
\mathit{srkind} & ::= & \mathsf{SharedRegion} \mid \mathit{srkn}\langle \mathit{owner}^*\rangle \\
\mathit{rkind} & ::= & \mathsf{Region} \mid \mathsf{NoGCRegion} \mid \mathsf{GCRegion} \mid \mathsf{LocalRegion} \mid \mathit{srkind} \\
k & ::= & \mathsf{Owner} \mid \mathsf{ObjOwner} \mid \mathit{rkind} \mid \ \mathit{rkind} : \mathsf{LT} \\
\\
\mathit{rpol} & ::= & \mathsf{LT}(\mathit{size}) \mid \mathsf{VT} \\
\mathit{tt} & ::= & \mathsf{NoRT} \mid \mathsf{RT} \\
\\
e & ::= & v \mid \mathsf{let}\ v = e\ \mathsf{in}\ \{\ e\ \} \mid v.\mathit{fd} \mid v.\mathit{fd} = v \mid v.\mathit{mn}\langle \mathit{owner}^*\rangle(v^*) \mid \mathsf{new}\ c \mid \\
 & & \mathsf{fork}\ v.\mathit{mn}\langle \mathit{owner}^*\rangle(v^*) \mid \mathsf{RT\_fork}\ v.\mathit{mn}\langle \mathit{owner}^*\rangle(v^*) \mid h.\mathit{fd} \mid h.\mathit{fd} = v \mid \\
 & & (\mathsf{RHandle}\langle r\rangle\ h)\ \{\ e\ \} \mid (\mathsf{RHandle}\langle \mathit{rkind} : \mathit{rpol}\ r\rangle\ h)\ \{\ e\ \} \mid (\mathsf{RHandle}\langle r\rangle\ h = [\mathsf{new}]_{\mathrm{opt}}\ h.\mathit{rsub})\ \{\ e\ \} \\
h & ::= & v \\
\\
\mathit{cn} & \in & \text{class names} \\
\mathit{fd} & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
\mathit{fn} & \in & \text{formal identifiers} \\
v, p & \in & \text{variable names} \\
r & \in & \text{region identifiers} \\
\mathit{srkn} & \in & \text{shared region kind names} \\
\mathit{rsub} & \in & \text{shared subregion names}
\end{array}
$$

Throughout this appendix, we try to use the same notations as in the grammar. However, to save space, we use o instead of *owner* and f instead of *formal*. We also use g and a to range over the set of owners. We assume the program source has been preprocessed by replacing each constraint "o_1 owns o_2" with the non-ASCII, but shorter form "$o_1 \succeq o_2$" and each constraint "o_1 outlives o_2" with "$o_1 \succeq_r o_2$."

We first define a number of predicates used in the type system. These are based on similar predicates from [70, 68]. Most of these predicates are straightforward and we omit their definitions for brevity; the only exception is $\mathit{InheritanceOK}(P)$, which we define formally in the set of rules.

Predicate	Meaning
$\mathit{WFClasses}(P)$	No class is defined twice and there is no cycle in the class hierarchy.
$\mathit{WFRegionKinds}(P)$	No region kind is defined twice and there is no cycle in the region kind hierarchy. In addition, no region kind has an infinite number of (transitive) subregions.
$\mathit{MembersOnce}(P)$	No class or region kind contains two fields with the same name, either declared or inherited. Similarly, no class declares two methods with the same name.
$\mathit{InheritanceOK}(P)$	The constraints of a sub-class/kind are included among the super-class/kind constraints. For overriding methods, in addition to the usual subtyping relations between the return/parameter types, the constraints of the overrider are included among the constraints of the overridden method; a similar relation holds for the effects.

The core of our type system is a set of typing judgments of the form $P; E; X; r_{cr} \vdash e : t$. P, the program being checked, is included to provide information about class definitions. The typing environment E provides information about the type of the free variables of e ($t\ v$, i.e., variable v has type t), the kind of the owners currently in scope ($k\ o$, i.e., owner o has kind k), and the two relations between owners: the "ownership" relation ($o_2 \succeq o_1$, i.e., o_2 owns o_1) and the "outlives" relation ($o_2 \succeq_r o_1$, i.e., o_2 $\mathsf{outlives}$ o_1). More formally, $E ::= \emptyset \mid E, t\ v \mid E, k\ o \mid E, o_2 \succeq o_1 \mid E, o_2 \succeq_r o_1$. r_{cr} is the current region. X must subsume the effects of e. t is the type of e.

We define the type system using the following judgments. We present the typing rules for these judgments after that.

Judgment	Meaning
$\vdash P : t$	Program P has type t.
$P \vdash_{\mathrm{def}} \mathit{defn}$	defn is a well formed class definition from program P.
$P \vdash_{\mathrm{srkdef}} \mathit{srkdef}_j$	srkdef_j is a well formed shared region kind definition from program P.
$P; E; X; r_{cr} \vdash e : t$	In program P, environment E, and current region r_{cr}, expression e has type t. Its evaluation accesses only objects (transitively) outlived by owners listed in the effects X.
$P \vdash_{\mathrm{env}} E$	E is a well formed environment with respect to program P.
$P; E \vdash_{\mathrm{meth}} \mathit{meth}$	Method definition meth is well defined with respect to program P and environment E.
$P \vdash \mathit{mbr} \in c$	Class c defines or inherits "member" definition mbr. "Member" refers to a field or a method: $\mathit{mbr} = \mathit{field} \mid \mathit{meth}$.
$P \vdash \mathit{rmbr} \in \mathit{rkind}$	Shared region kind rkind defines or inherits "region member" definition rmbr. "Region member" refers to a field or a subregion: $\mathit{rmbr} = \mathit{field} \mid \mathit{subsreg}$.
$P; E \vdash_{\mathrm{type}} t$	t is a well formed type with respect to program P and environment E.
$P \vdash t_1 \leq t_2$	In program P, t_1 is a subtype of t_2.
$P; E \vdash_{\mathrm{okind}} k$	k is a well formed owner kind, with respect to program P and environment E.
$P \vdash k_1 \leq_{\mathrm{k}} k_2$	In program P, k_1 is a subkind of k_2.
$E \vdash X_2 \succeq_r X_1$	Effect X_1 is subsumed by effect X_2, with respect to environment E.
$E \vdash \mathit{constr}$	In environment E, constr is well formed (i.e., the owners involved in it are well formed) and satisfied.
$E \vdash_{\mathrm{k}} o : k$	In environment E, o is a well formed owner with kind k.
$E \vdash \mathrm{RKind}(o) = k$	Owner o is either a region of kind k or an object allocated in such a region.
$E \vdash_{\mathrm{av}} \mathsf{RH}(o)$	The handle of the region o is allocated in (if o is an object) or o stands for (if it is a region) is available in the environment E.
$E \vdash o_2 \succeq o_1$	In environment E, owner o_2 (an object or a region) owns owner o_1 (which must be an object).
$E \vdash o_2 \succeq_r o_1$	In environment E, owner o_2 outlives owner o_1.

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses}(P) \quad \mathit{WFRegionKinds}(P) \quad \mathit{MembersOnce}(P) \quad \mathit{InheritanceOK}(P) \\ P = \mathit{defn}_{1..n}\ \mathit{srkdef}_{1..r}\ e \quad \forall i = 1..n,\ P \vdash_{\text{def}} \mathit{defn}_i \quad \forall i = 1..r,\ P \vdash_{\text{srkdef}} \mathit{srkdef}_j \\ E = \ \emptyset, \mathsf{GCRegion\ heap}, \mathsf{SharedRegion\!:\!LT\ immortal}, \mathsf{RHandle}\langle\mathsf{heap}\rangle\ \mathsf{h}_{\mathsf{heap}}, \mathsf{RHandle}\langle\mathsf{immortal}\rangle\ \mathsf{h}_{\mathsf{immortal}} \\ P; E; \mathsf{world}; \mathsf{heap} \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash_{\text{def}} \mathit{defn}}$

[CLASS DEF]

$$\frac{\begin{array}{c} \begin{array}{rl} E = & \emptyset, \mathsf{GCRegion\ heap}, \mathsf{SharedRegion\!:\!LT\ immortal}, \mathsf{RHandle}\langle\mathsf{heap}\rangle\ \mathsf{h}_{\mathsf{heap}}, \mathsf{RHandle}\langle\mathsf{immortal}\rangle\ \mathsf{h}_{\mathsf{immortal}}, \\ & (k_i\ \mathit{fn}_i)_{i=1..n}, \mathit{constr}_{1..p}, \mathit{cn}\langle \mathit{fn}_{1..n}\rangle\ \mathsf{this}, (\mathit{fn}_i \succeq_r \mathit{fn}_1)_{i=2..n} \end{array} \\ \mathit{defn} = \mathsf{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i=1..n}\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ \mathit{constr}_{1..p}\ \{\ \mathit{field}_{1..m}\ \mathit{meth}_{1..q}\ \} \quad P = \ldots \mathit{defn} \ldots \\ P \vdash_{\text{env}} E \quad P; E \vdash_{\text{type}} c \quad \forall j = 1..m,\ (\mathit{field}_j = t_j\ \mathit{fd}_j\ \wedge\ P; E \vdash_{\text{type}} t_j) \quad \forall k = 1..q,\ P; E \vdash_{\text{meth}} \mathit{meth}_k \end{array}}{P \vdash_{\text{def}} \mathit{defn}}$$

$\boxed{P \vdash_{\text{srkdef}} \mathit{srkdef}}$

[REGION KIND DEF]

$$\frac{\begin{array}{c} \mathit{srkdef}\ = \mathsf{regionKind}\ \mathit{srkn}\langle (k_i\ \mathit{fn}_i)_{i=1..n}\rangle\ \mathsf{extends}\ r\ \mathsf{where}\ \mathit{constr}_{1..c}\ \{\ \mathit{field}_{1..m}\ \mathit{subsreg}_{1..s}\ \} \quad P = \ldots\ \mathit{srkdef}\ \ldots \\ E = \emptyset, (k_i\ \mathit{fn}_i)_{i=1..n}, \mathit{constr}_{1..c}, \mathit{srkn}\langle \mathit{fn}_{1..n}\rangle\ \mathsf{this}, (\mathit{fn}_i \succeq_r \mathsf{this})_{i=1..n} \quad P \vdash_{\text{env}} E \quad P; E \vdash_{\text{okind}} r \\ \forall j = 1..m,\ (\mathit{field}_j = t_j\ \mathit{fd}_j\ \wedge\ P; E \vdash_{\text{type}} t_j) \\ \forall k = 1..s,\ (\mathit{subsreg}_k = \mathit{srkind}_k\!:\!\mathit{rpol}_k\ \mathit{tt}\ \mathit{rsub}_k\ \wedge\ P; E \vdash_{\text{srkind}} \mathit{srkind}_k) \end{array}}{P \vdash_{\text{srkdef}} \mathit{srkdef}}$$

$\boxed{P \vdash_{\text{env}} E}$

[ENV $\emptyset$]

$$\frac{}{P \vdash_{\text{env}} \emptyset}$$

[ENV v]

$$\frac{\begin{array}{c} P \vdash_{\text{env}} E \\ v \notin \mathit{Dom}(E) \\ P; E \vdash_{\text{type}} t \end{array}}{P \vdash_{\text{env}} E,\ t\ v}$$

[ENV OWNER]

$$\frac{\begin{array}{c} P \vdash_{\text{env}} E \\ o \notin \mathit{Dom}(E) \\ P; E \vdash_{\text{okind}} k \end{array}}{P \vdash_{\text{env}} E,\ k\ o}$$

[ENV $\succeq$]

$$\frac{\begin{array}{c} P \vdash_{\text{env}} E \\ E \vdash_{\text{k}} o_1 : \mathsf{ObjOwner} \\ E \vdash_{\text{k}} o_2 : k \end{array}}{P \vdash_{\text{env}} E,\ o_1 \succeq o_2}$$

[ENV $\succeq_r$]

$$\frac{\begin{array}{c} P \vdash_{\text{env}} E \\ E \vdash_{\text{k}} o_1 : k_1 \\ E \vdash_{\text{k}} o_2 : k_2 \end{array}}{P \vdash_{\text{env}} E,\ o_1 \succeq_r o_2}$$

$\boxed{P; E \vdash_{\text{type}} t}$

[TYPE INT]

$$\frac{}{P; E \vdash_{\text{type}} \mathsf{int}}$$

[TYPE OBJECT]

$$\frac{E \vdash_{\text{k}} o : k}{P; E \vdash_{\text{type}} \mathsf{Object}\langle o\rangle}$$

[TYPE REGION HANDLE]

$$\frac{\begin{array}{c} E \vdash_{\text{k}} r : k \\ P \vdash k \leq_{\text{k}} \mathsf{Region} \end{array}}{P; E \vdash_{\text{type}} \mathsf{RHandle}\langle r\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} P = \ldots\ \mathit{defn}\ \ldots \quad \mathit{defn} = \mathsf{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i=1..n}\rangle\ \ldots\ \mathsf{where}\ \mathit{constr}_{1..c}\ \ldots \\ \forall i = 1..n,\ (E \vdash_{\text{k}} o_i : k'_i\ \wedge\ P \vdash k'_i \leq_{\text{k}} k_i\ \wedge\ E \vdash o_i \succeq_r o_1) \quad \forall i = 1..c,\ E \vdash \mathit{constr}_i[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n] \end{array}}{P; E \vdash_{\text{type}} \mathit{cn}\langle o_{1..n}\rangle}$$

$\boxed{P \vdash t_1 \leq t_2}$

[SUBTYPE REFL]

$$\frac{}{P \vdash t \leq t}$$

[SUBTYPE TRANS]

$$\frac{P \vdash t_1 \leq t_2 \quad P \vdash t_2 \leq t_3}{P \vdash t_1 \leq t_3}$$

[SUBTYPE CLASS]

$$\frac{P \vdash_{\text{def}}\ \ \mathsf{class}\ \mathit{cn}\langle (k_i\ \mathit{fn}_i)_{i=1..n}\rangle \mathsf{extends}\ \mathit{cn}_2\langle \mathit{fn}_1\ o^*\rangle\ \ldots}{P \vdash \mathit{cn}\langle o_{1..n}\rangle \leq \mathit{cn}_2\langle \mathit{fn}_1\ o^*\rangle[o_1/\mathit{fn}_1]..[o_n/\mathit{fn}_n]}$$

$\boxed{P;E \vdash_{\mathrm{meth}} meth}$

[METHOD]

$$\frac{\begin{array}{c} E' = E, f_{1..n}, constr_{1..c}, (t_j\ \ p_j)_{j=1..p}, \mathsf{Region\ initialRegion}, \mathsf{RHandle}\langle\mathsf{initialRegion}\rangle\ h_{fresh} \\ P \vdash_{\mathrm{env}} E' \forall i = 1..q,\ (E' \vdash_{\mathrm{k}} a_i : k_i\ \vee\ a_i = \mathsf{RT}) \qquad P;E';a_{1..q};\mathsf{initialRegion} \vdash e : t \end{array}}{P;E \vdash_{\mathrm{meth}} t\ mn\langle f_{1..n}\rangle((t_j\ \ p_j)_{j=1..p})\ \mathsf{accesses}\ a_{1..q}\ \mathsf{where}\ constr_{1..c}\ \{e\}}$$

$\boxed{P;E \vdash_{\mathrm{okind}} k}$

[STANDARD OWNERS]

$$\frac{k \in \{\mathsf{Owner}, \mathsf{ObjOwner}, \mathsf{Region}, \mathsf{GCRegion}, \mathsf{NoGCRegion}, \mathsf{LocalRegion}, \mathsf{SharedRegion}\}}{P;E \vdash_{\mathrm{okind}} k}$$

[LT REGIONS]

$$\frac{P;E \vdash_{\mathrm{okind}} rkind}{P;E \vdash_{\mathrm{okind}} rkind : \mathsf{LT}}$$

[USER DECLARED SHARED REGION]

$$\frac{\begin{array}{c} P \vdash_{\mathrm{srkdef}} \mathsf{regionKind}\ srkn\langle(k_i\ fn_i)_{i=1..n}\rangle\ \dots\ \mathsf{where}\ constr_{1..c}\ \dots \\ \forall i = 1..n,\ (E \vdash_{\mathrm{k}} o_i : k_i' \quad P \vdash k_i' \leq_{\mathrm{k}} k_i) \\ \forall i = 1..c,\ E \vdash constr_i[o_1/fn_1]..[o_n/fn_n] \end{array}}{P;E \vdash_{\mathrm{okind}} srkn\langle o_{1..n}\rangle}$$

$\boxed{P \vdash k_1 \leq_{\mathrm{k}} k_2}$

[SUBKIND REFL]

$$\frac{}{P \vdash k \leq_{\mathrm{k}} k}$$

[SUBKIND TRANS]

$$\frac{P \vdash k_1 \leq_{\mathrm{k}} k_2 \qquad P \vdash k_2 \leq_{\mathrm{k}} k_3}{P \vdash k_1 \leq_{\mathrm{k}} k_3}$$

[SUBKIND VALUE]

$$\frac{}{P \vdash \mathsf{Value} \leq_{\mathrm{k}} k}$$

[SUBKIND OWNER]

$$\frac{k \in \{\mathsf{ObjOwner}, \mathsf{Region}\}}{P \vdash k \leq_{\mathrm{k}} \mathsf{Owner}}$$

[SUBKIND REGION]

$$\frac{k \in \{\mathsf{GCRegion}, \mathsf{NoGCRegion}\}}{P \vdash k \leq_{\mathrm{k}} \mathsf{Region}}$$

[SUBKIND NOGCREGION]

$$\frac{k \in \{\mathsf{LocalRegion}, \mathsf{SharedRegion}\}}{P \vdash k \leq_{\mathrm{k}} \mathsf{NoGCRegion}}$$

[SUBKIND SHARED REGION KIND]

$$\frac{P \vdash_{\mathrm{srkdef}} \mathsf{regionKind}\ srkn\langle(k_i\ fn_i)_{i=1..n}\rangle\ \mathsf{extends}\ r\ \dots}{P \vdash srkn\langle o_1..o_n\rangle \leq_{\mathrm{k}} r[o_1/fn_1]..[o_n/fn_n]}$$

[DELETE LT]

$$\frac{}{P \vdash rkind : \mathsf{LT} \leq_{\mathrm{k}} rkind}$$

[ADD LT]

$$\frac{P \vdash rkind_1 \leq_{\mathrm{k}} rkind_2}{P \vdash rkind_1 : \mathsf{LT} \leq_{\mathrm{k}} rkind_2 : \mathsf{LT}}$$

$\boxed{P \vdash mbr \in c, \text{ where } mbr = field \mid meth}$

[INHERITED CLASS MEMBER]

$$\frac{P \vdash_{\mathrm{def}} \mathsf{class}\ cn_2\langle(k_i\ fn_i')_{i=1..m}\rangle\ \mathsf{extends}\ cn\langle o_{1..n}\rangle \dots \qquad P \vdash mbr \in cn\langle fn_{1..n}\rangle}{P \vdash mbr[o_1/fn_1]..[o_n/fn_n] \in cn_2\langle fn_{1..m}'\rangle}$$

[DECLARED CLASS MEMBER]

$$\frac{P \vdash_{\mathrm{def}} \mathsf{class}\ cn\langle(k_i\ fn_i)_{i=1..n}\rangle\ \dots\ \{\dots\ mbr\ \dots\}}{P \vdash mbr \in cn\langle fn_{1..n}\rangle}$$

$\boxed{P \vdash rmbr \in rkind, \text{ where } rmbr = field \mid subsreg}$

[DECLARED REGION MEMBER]

$$\frac{P \vdash_{\mathrm{srkdef}} \mathsf{regionKind}\ srkn\langle(k_i\ fn_i)_{i=1..n}\rangle\ \dots\ \{\ \dots\ rmbr\ \dots\ \}}{P \vdash rmbr \in srkn\langle fn_{1..n}\rangle}$$

[INHERITED REGION MEMBER]

$$\frac{\begin{array}{c} P \vdash rmbr \in srkn\langle fn_{1..n}\rangle \\ P \vdash_{\mathrm{srkdef}} \mathsf{regionKind}\ srkn\langle(k_i\ fn_i')_{i=1..m}\rangle\ \mathsf{extends}\ srkn\langle o_{1..n}\rangle\ \dots \end{array}}{P \vdash rmbr[o_1/fn_1]..[o_n/fn_n] \in srkn_2\langle fn_{1..m}'\rangle}$$

$\boxed{E \vdash constr}$

[ENV CONSTR]

$$\frac{E = E_1, constr, E_2}{E \vdash constr}$$

[$\succeq$ world]

$$\frac{o \neq \mathsf{RT}}{E \vdash \mathsf{world} \succeq_r o}$$

[$\succeq$ OWNER]

$$\frac{E = E_1,\ cn\langle o_{1..n}\rangle\ \mathsf{this},\ E_2}{E \vdash o_1 \succeq \mathsf{this}}$$

[$\succeq$ REFL]

$$\frac{}{E \vdash o \succeq o}$$

[$\succeq$ TRANS]

$$\frac{E \vdash o_1 \succeq o_2 \qquad E \vdash o_2 \succeq o_3}{E \vdash o_1 \succeq o_3}$$

[$\succeq\ \rightarrow\ \succeq_r$]

$$\frac{E \vdash o_1 \succeq o_2}{E \vdash o_1 \succeq_r o_2}$$

[$\succeq_r$ heap/immortal]

$$\frac{o_1 \in \{\mathsf{heap}, \mathsf{immortal}\}}{E \vdash o_1 \succeq_r o_2}$$

$\boxed{E \vdash X_1 \succeq_r X_2}$ $\boxed{E \vdash_{\mathrm{k}} o : k}$

[$\succeq_r$ REFL]

$$\frac{}{E \vdash o \succeq_r o}$$

[$\succeq_r$ TRANS]

$$\frac{E \vdash o_1 \succeq_r o_2 \qquad E \vdash o_2 \succeq_r o_3}{E \vdash o_1 \succeq_r o_3}$$

[$X_1 \succeq_r X_2$]

$$\frac{\forall o \in X_1,\, \exists g \in X_2,\, E \vdash o \succeq_r g}{E \vdash X_1 \succeq_r X_2}$$

[OWNER THIS]

$$\frac{E = E_1, cn\langle ...\rangle\ \mathsf{this}, E_2}{E \vdash_{\mathrm{k}} \mathsf{this} : \mathsf{Owner}}$$

[OWNER FORMAL]

$$\frac{E = E_1, k\ o, E_2}{E \vdash_{\mathrm{k}} o : k}$$

$\boxed{E \vdash \mathrm{RKind}(o) = k}$

[RKIND THIS]

$$\frac{\begin{array}{c} E = E_1, cn\langle o_{1..n}\rangle\ \mathsf{this}, E_2 \\ E \vdash \mathrm{RKind}(o_1) = k \end{array}}{E \vdash \mathrm{RKind}(\mathsf{this}) = k}$$

[RKIND FN1]

$$\frac{\begin{array}{c} E \vdash_{\mathrm{k}} fn : k \\ k \notin \{\mathsf{Owner}, \mathsf{ObjOwner}\} \end{array}}{E \vdash \mathrm{RKind}(fn) = k}$$

[RKIND FN2]

$$\frac{\begin{array}{c} E \vdash_{\mathrm{k}} fn : k \qquad k \in \{\mathsf{Owner}, \mathsf{ObjOwner}\} \\ E \vdash o \succeq fn \qquad E \vdash \mathrm{RKind}(o) = k_2 \end{array}}{E \vdash \mathrm{RKind}(fn) = k}$$

$\boxed{E \vdash_{\mathrm{av}} \mathsf{RH}(o)}$

[AV HANDLE]

$$\frac{E = E_1, \mathsf{RHandle}\langle r\rangle\ h, E_2}{E \vdash_{\mathrm{av}} \mathsf{RH}(r)}$$

[AV THIS]

$$\frac{E = E_1, cn\langle o_{1..n}\rangle\ \mathsf{this}, E_2}{E \vdash_{\mathrm{av}} \mathsf{RH}(\mathsf{this})}$$

[AV TRANS1]

$$\frac{E \vdash o_1 \succeq o_2 \qquad E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}$$

[AV TRANS2]

$$\frac{E \vdash o_2 \succeq o_1 \qquad E \vdash_{\mathrm{av}} \mathsf{RH}(o_2)}{E \vdash_{\mathrm{av}} \mathsf{RH}(o_1)}$$

$\boxed{InheritanceOK(P)}$

[INHERITANCEOK REGION KIND]

$$\frac{\begin{array}{c} srkdef\ = \mathsf{regionKind}\ srkn\langle f_{1..n}\rangle\ \mathsf{extends}\ srkn'\langle o_{1..m}\rangle\ \mathsf{where}\ constr_{1..q}\ ... \\ srkdef' = \mathsf{regionKind}\ srkn'\langle (k'_i\ fn'_i)_{i=1..m}\rangle\ \mathsf{extends}\ srkind\ \mathsf{where}\ constr'_{1..s}\ ... \\ P \vdash_{\mathrm{srkdef}} srkdef' \qquad constr_{1..s}[o_1/fn'_1]..[o_m/fn'_m] \subseteq constr'_{1..q} \end{array}}{P \vdash InheritanceOK(srkdef)}$$

[INHERITANCEOK PROG]

$$\frac{\begin{array}{c} P = defn_{1..n}\ srkdef_{1..r}\ e \\ \forall i = 1..n,\, P \vdash InheritanceOK(defn_i) \\ \forall i = 1..r,\, P \vdash InheritanceOK(srkdef_i) \end{array}}{InheritanceOK(P)}$$

[INHERITANCEOK CLASS]

$$\frac{\begin{array}{l} defn = \mathsf{class}\ cn\langle (k_i\ fn_i)_{i=1..n}\rangle\ \mathsf{extends}\ cn\langle o_{1..m}\rangle \mathsf{where}\ constr_{1..q}\ ... \\ defn' = \mathsf{class}\ cn'\langle (k'_i\ fn'_i)_{i=1..m}\rangle\ \mathsf{extends}\ c\ \mathsf{where}\ constr'_{1..u}\ ... \\ \quad P \vdash_{\mathrm{def}} defn' \qquad constr_{1..u}[o_1/fn'_1]..[o_m/fn'_m] \subseteq constr'_{1..q} \\ \forall mn, \quad (\quad P \vdash meth \in defn \quad \wedge \quad meth = t_r\ mn\langle ...\rangle(...)\ ... \quad \wedge \\ \qquad\qquad\quad P \vdash meth' \in defn' \quad \wedge \quad meth = t'_r\ mn\langle ...\rangle(...)\ ... \quad) \\ \qquad \rightarrow \quad P \vdash OverridesOK(meth, meth') \end{array}}{P \vdash InheritanceOK(defn)}$$

[OVERRIDESOK METHOD]

$$\frac{\begin{array}{c} meth = \ t_r\ mn\langle f_{1..n}\rangle((t_i\ p_i)_{i=1..m})\ \mathsf{accesses}\ a_{1..q} \mathsf{where}\ constr_{1..r}\ ... \\ meth' = \ t'_r\ mn\langle f_{1..n}\rangle((t'_i\ p'_i)_{i=1..m})\ \mathsf{accesses}\ a'_{1..s} \mathsf{where}\ constr'_{1..u}\ ... \\ P \vdash t'_r \leq t_r \qquad \forall i = 1..m,\, P \vdash t_i \leq t'_i \\ a'_{1..q} \subseteq a_{1..s} \qquad constr'_{1..r} \subseteq constr_{1..u} \end{array}}{P \vdash OverridesOK(meth, meth')}$$

$\boxed{P; E; X; r_{cr} \vdash e : t}$

[EXPR VAR]

$$\frac{E = E_1, t\ v, E_2}{P; E; X; r_{cr} \vdash v : t}$$

[EXPR REF READ]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \qquad t' = t[o_1/fn_1]..[o_n/fn_n] \\ (t' = \mathsf{int})\ \vee\ (t' = cn'\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq_r o'_1) \end{array}}{P; E; X; r_{cr} \vdash v.fd : t'}$$

[EXPR REf WRITE]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v_1 : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \qquad t' = t[o_1/fn_1]..[o_n/fn_n] \\ P; E; X; r_{cr} \vdash v_2 : t_2 \qquad P \vdash t_2 \leq t' \\ (t' = \mathsf{int})\ \vee\ (t' = cn'\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq_r o'_1) \end{array}}{P; E; X; r_{cr} \vdash v_1.fd = v_2 : t'}$$

[EXPR INVOKE]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash v : cn\langle o_{1..n}\rangle \quad \forall i=(n+1)..m,\ E \vdash o_i \succeq_r o_1 \quad P \vdash meth \in cn\langle fn_{1..n}\rangle \\ meth =\ t\ mn\langle (k_i\ fn_i)_{i=(n+1)..m}\rangle\ ((t_j\ p_j)_{j=1..u})\mathsf{accesses}\ X_m\ \mathsf{where}\ constr_{1..c}\ \{e\} \\ Rename(\alpha) \stackrel{def}{=} \alpha[o_1/fn_1]..[o_m/fn_m][r_{cr}/\mathsf{initialRegion}] \\ \forall i=1..u,\ (P;E;X;r_{cr} \vdash v_i : t'_i\ \wedge\ P \vdash t'_i \leq Rename(t_i)) \\ \forall i=(n+1)..m,\ (E \vdash_{\mathrm{k}} o_i : k'_i\ \wedge\ P \vdash k'_i \leq_{\mathrm{k}} Rename(k_i)) \\ E \vdash X \succeq_r Rename(X_m) \quad \forall i=1..c,\ E \vdash Rename(constr_i) \end{array}}{P;E;X;r_{cr} \vdash v.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : Rename(t)}$$

[EXPR LET]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash e_1 : t_1 \quad E' = E, t_1\ v \\ P \vdash_{\mathrm{env}} E' \quad P;E';X;r_{cr} \vdash e_2 : t_2 \end{array}}{P;E;X;r_{cr} \vdash \mathsf{let}\ v = e_1\ \mathsf{in}\ e_2 : t_2}$$

[EXPR NEW]

$$\frac{\begin{array}{c} P;E \vdash_{\mathrm{type}} c \quad c = cn\langle o_{1..n}\rangle \\ E \vdash_{\mathrm{av}} \mathsf{RH}(o_1) \quad E \vdash X \succeq_r o_1 \end{array}}{P;E;X;r_{cr} \vdash \mathsf{new}\ c : c}$$

[EXPR REGION]

$$\frac{\begin{array}{c} P;E \vdash_{\mathrm{okind}} rkind \quad rkind = srkn\langle\rangle \\ k_r = \begin{cases} rkind : \mathsf{LT} & \text{if } rpol = \mathsf{LT}(size) \\ rkind & \text{otherwise} \end{cases} \\ E_2 = E, k_r\ r, \mathsf{RHandle}\langle r\rangle\ h \\ E_3 = \begin{cases} E_2, (r_e \succeq_r r)_{\forall r_e \in Regions(E)} & \\ \quad \text{if} P \vdash rkind \leq_{\mathrm{k}} \mathsf{LocalRegion} & \\ E_2 \quad \text{otherwise} & \end{cases} \\ P \vdash_{\mathrm{env}} E_3 \quad P;E_3;X,r;r \vdash e : t \quad E \vdash X \succeq_r \mathsf{heap} \end{array}}{P;E;X;r_{cr} \vdash (\mathsf{RHandle}\langle rkind : rpol\ \ r\rangle\ h)\ \{e\} : \mathsf{int}}$$

[EXPR SUBREGION]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash h_2 : \mathsf{RHandle}\langle r_2\rangle \quad E \vdash_{\mathrm{k}} r_2 : srkn_2\langle o_{1..n}\rangle \\ P \vdash rkind_3 : rpol\ \ tt\ \ rsub \in srkn_2\langle fn_{1..n}\rangle \\ rkind = rkind_3[o_1/fn_1]..[o_n/fn_n][r_2/\mathsf{this}] \\ k_r = \begin{cases} rkind : \mathsf{LT} & \text{if } rpol = \mathsf{LT}(size) \\ rkind & \text{otherwise} \end{cases} \\ E_2 = E, k_r\ r, \mathsf{RHandle}\langle r\rangle\ h, r_2 \succeq_r r \quad P \vdash_{\mathrm{env}} E_2 \\ P;E_2;X,r;r \vdash e : t \\ (\mathsf{new}\ \vee\ (rpol = VT)\ \vee\ (tt = \mathsf{NoRT}))\ \rightarrow\ E \vdash X \succeq_r \mathsf{heap} \\ (tt = \mathsf{RT})\ \rightarrow\ E \vdash X \succeq_r \mathsf{RT} \end{array}}{P;E;X;r_{cr} \vdash (\mathsf{RHandle}\langle r\rangle\ h_1 = [\mathsf{new}]_{\mathrm{opt}}\ h_2.rsub)\ \{e\} : \mathsf{int}}$$

$Regions(E)$ is the set of all regions in E, defined as follows:

$$\begin{array}{lcl} Regions(\emptyset) & = & \emptyset \\ Regions(E, rkind\ \ r) & = & Regions(E) \cup \{r\} \\ Regions(E, \_) & = & Regions(E), \text{ otherwise} \end{array}$$

[EXPR LOCAl REGION]

$$\frac{P;E;X;r_{cr} \vdash (\mathsf{RHandle}\langle \mathsf{LocalRegion} : \mathsf{VT}\ \ r\rangle\ h)\ \{e\} : \mathsf{int}}{P;E;X;r_{cr} \vdash (\mathsf{RHandle}\langle r\rangle\ h)\ \{e\} : \mathsf{int}}$$

[EXPR FORK]

$$\frac{\begin{array}{c} NonLocal(k) \stackrel{def}{=} (P \vdash k \leq_{\mathrm{k}} \mathsf{SharedRegion})\ \vee\ (P \vdash k \leq_{\mathrm{k}} \mathsf{GCRegion}) \\ E \vdash \mathrm{RKind}(r_{cr}) = k_{cr} \quad NonLocal(k_{cr}) \quad P;E;X;r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i=1..m,\ (E \vdash \mathrm{RKind}(o_i) = k_i\ \wedge\ NonLocal(k_i)) \quad P;E;X \setminus \{\mathsf{RT}\};r_{cr} \vdash v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \end{array}}{P;E;X;r_{cr} \vdash \mathsf{fork}\ v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \mathsf{int}}$$

[EXPR RTFORK]

$$\frac{\begin{array}{c} X' = \{o \in X \mid E \vdash \mathrm{RKind}(o) = k\ \wedge\ P \vdash k \leq_{\mathrm{k}} \mathsf{SharedRegion} : \mathsf{LT}\} \\ P;E;X',\mathsf{RT};r_{cr} \vdash v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \quad P;E;X;r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i=1..m,\ (E \vdash \mathrm{RKind}(o_i) = k_i\ \wedge\ P \vdash k_i \leq_{\mathrm{k}} \mathsf{SharedRegion}) \quad E \vdash \mathrm{RKind}(r_{cr}) = k_{cr} \quad P \vdash k_{cr} \leq_{\mathrm{k}} \mathsf{SharedRegion} \end{array}}{P;E;X;r_{cr} \vdash \mathsf{RT\_fork}\ v_0.mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \mathsf{int}}$$

[EXPR GET REGION FIELD]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash h : \mathsf{RHandle}\langle r\rangle \quad E \vdash_{\mathrm{k}} r : srkn\langle o_{1..n}\rangle \quad P \vdash t\ fd \in srkn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n][r/\mathsf{this}] \quad ((t' = \mathsf{int})\ \vee\ (t' = cn\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq_r o'_1)) \end{array}}{P;E;X;r_{cr} \vdash h.fd : t'}$$

[EXPR SET REGION FIELD]

$$\frac{\begin{array}{c} P;E;X;r_{cr} \vdash h : \mathsf{RHandle}\langle r\rangle \quad E \vdash_{\mathrm{k}} r : srkn\langle o_{1..n}\rangle \quad P \vdash t\ fd \in srkn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n][r/\mathsf{this}] \quad P;E;X;r_{cr} \vdash v : t_1 \quad P \vdash t_1 \leq t' \\ ((t' = \mathsf{int})\ \vee\ (t' = cn\langle o'_{1..m}\rangle\ \wedge\ E \vdash X \succeq_r o'_1)) \end{array}}{P;E;X;r_{cr} \vdash h.fd = v : t'}$$

6.B Translation to RTSJ

This sections presents details on how we translate SafeJava programs written to RTSJ programs using local translation rules. The following subsections show the translation for expressions that create or enter regions, access region fields, allocate objects, or start threads. The translation for the other language constructs is trivial—we do a simple type-erasure based translation. In this section, we use bold font for the generated code and *italic font* (default font for mathematical notation in LaTeX) for the expressions that are evaluated at translation time.

6.B.1 Region Representation

Similar to SafeJava, RTSJ allows the programmers to create memory regions ("memory areas" in RTSJ terminology). It also has two special memory areas: a garbage-collected heap and an immortal memory area. A memory area is a normal Java object that also points to a piece of memory for the objects allocated in that memory area. Figure 6-16 presents the RTSJ hierarchy of classes for memory management. The root of this hierarchy, MemoryArea, is subclassed by HeapMemory, ImmortalMemory and ScopedMemory. HeapMemory and ImmortalMemory represent the heap and the immortal memory area respectively; there exists only one instance of each of them. ScopedMemory is the class for normal regions; it has two subclasses: LTMemory (for LT regions) and VTMemory (for VT regions). The RTSJ regions are maintained similarly to our shared regions: each thread has a stack of regions it can currently access, and each region has a counter of how many threads can access it.

We represent a region r from SafeJava as an RTSJ memory area m plus two auxiliary objects w1 and w2. m points to a piece of memory that is pre-allocated for an LT region, or grown on-demand for a VT region. m also points to an object w1 whose fields point to the representation of r's subregions (we subclass LT/VTMemory to add an extra field); these subregions are represented in the same way as r. (Note that we only allow a region to have a bounded number of subregions.) In addition, m's portal points to an object w2 that serves as a wrapper for r's fields. w2 is allocated in the memory space attached to m, while m, w1, and all the similar objects used for the representation of r's transitive subregions are allocated in the region that was the current region at the time m was created. Figure 6-12 from Section 6.6 presents the translation of a region with three fields and two subregions into RTSJ.

There are several differences between the RTSJ memory areas and the SafeJava regions. The following list presents them, together with our translation for each case; Figure 6-15 presents the classes and interfaces we introduce along the way.

1. An RTSJ memory area has one portal field, similar to a SafeJava region field, except that it is untyped (i.e., it has type Object). To allow multiple and typed region fields, for each region kind *rkind* from SafeJava, we introduce a *field wrapper* class *rkind*Fields (Figure 6-15) that contains a field for each original field. For each region of kind *rkind*, the portal of the corresponding memory area points to an object of class *rkind*Fields allocated in that memory area.

2. RTSJ does not have anything equivalent to the subregions in SafeJava. To simulate them, for each region kind *rkind*, we introduce a *subregion wrapper* class *rkind*Subs

```
public interface Irkind extends IRegion {
  rkindSubs getSubs();
  rkindFields getFields();
}

public class rkindFields {
  ∀ field (t fd) ∈ rkind
    public t fd;
}

public class rkindSubs {
  ∀ subregion (srkind_s:rpol_s rsub) ∈ rkind
    public Isrkind_s rsub;
}


public interface IRegion {
  boolean isFlushed();

  void enterRegion();
  void exitRegion();

  boolean isASubregion();
  void    setIsASubregion(boolean value);
}
```

Figure 6-15: Declarations of I*rkind* and IRegion

(Figure 6-15) that has one field for each subregion. Subregions are represented similarly to their parent region. When we create a region, we automatically create all its transitive subregions. All the memory area objects and the field wrappers are allocated in the regions that is the current region at the creation time.

3. Each memory area that represents a region of kind *rkind* implements the interface I*rkind* (Figure 6-15); this interface has special methods for retrieving the field wrapper object and the subregion wrapper. In addition, I*rkind* extends the interface IRegion (Figure 6-15) that has some region maintenance methods that are explained later in this appendix.

4. To ensure proper nesting of memory area enter/exit operations, RTSJ uses the following mechanism for executing code inside a memory area: the program calls the enter method of the memory area object and passes it a Runnable object; the run() method of this object is executed inside the memory area. To translate an expression of the form "(RHandle$\langle rkind : rpol\ \ r\rangle$ h) $\{e\}$" into this pattern, we have to create a Runnable object.

5. RTSJ does not have thread-local memory areas. Therefore, we have to translate the local regions into memory areas that are maintained through thread counting, even if we know that only one thread uses them. This is less efficient than a genuine implementation of local regions but is still correct.

6. Our RTSJ platform flushes a memory area when its counter changes from one to zero and its portal is null. While designing our language, our goal was as follows. We

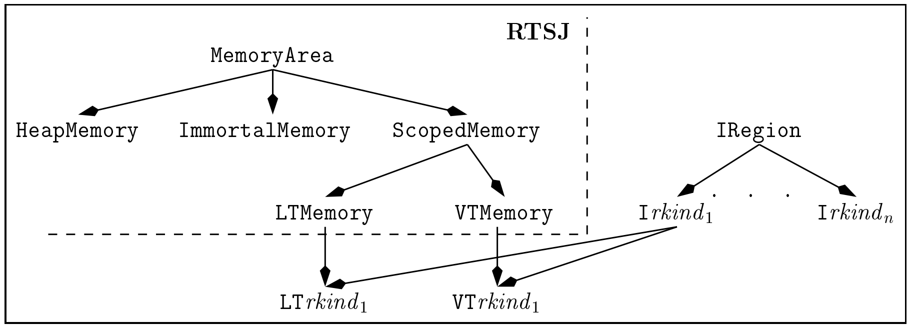

Figure 6-16: RTSJ Hierarchy of Classes for Memory Management and our Extensions to it

wanted to provide semantics where flushing or deleting of regions is transparent to programs (so one cannot detect under program control if a region has been flushed or deleted). However, we also wanted to reclaim memory space as soon as possible. We therefore came up with the following rules for flushing a region:

- we flush a subregion if the counter is zero, all fields are null and all subregions have been flushed;
- we flush a region as soon as its counter is zero (because the region won't be used afterward).

To enforce our rules, at each place where we might exit a region we call its method exitRegion(). If the above flushing conditions are satisfied, this method sets the portal field to null such that RTSJ flushes that region; otherwise, the portal field remains non-null and prevents the region from being flushed. To check the flushing conditions, we use the methods isFlushed() and isASubregion().

In RTSJ, the programmer chooses the desired allocation policy by allocating a LTMemory or a VTMemory object. Hence, for each region kind *rkind*, we define two implementations of I*rkind*: a class LT*rkind* that extends LTMemory, and a class VT*rkind* that extends VTMemory. This creates the class hierarchy from Figure 6-16. Figure 6-17 presents the declaration of LT*rkind*, except for the implementations of enterRegion() and exitRegion() that are presented later in this appendix. VT*rkind* is almost identical, except that it inherits from VTMemory, and its constructor does not take any size argument. As Java does not have multiple class inheritance, there is significant code duplication between LT*rkind* and VT*rkind*. This can be improved by factoring out most of the code as static methods in one of the two classes.

The private method extraFlushingTest() tests the flushing conditions that are required *in addition to* the zero-valued counter: top-level region or subregion with all portal fields null and all subregions flushed. The private method flush() creates the condition for the flushing of the region by the RTSJ platform: sets the portal field to null. In the case of a top-level region, it also applies itself recursively to all transitive subregions. Together, these two methods implement the flushing policy described above; they are used by exitRegion() (presented in Section 6.B.3).

In SafeJava, we have both region names and region handles. The region names are for typechecking purposes only and are removed by the type erasure. The region handles are runtime values; a region handle that had the type $\mathsf{RHandle}\langle r\rangle$ in SafeJava is translated into a reference to an object $\mathsf{I}\mathit{rkind}$, where *rkind* is the kind of the region r.

6.B.2 Creating a Default Local Region

Instead of presenting directly the translation for general region creation expression of the form "$(\mathsf{RHandle}\langle \mathit{rkind}:\mathit{rpol}\ r\rangle\ h)\ \{e\}$", we first look at a simplified case. An expression of the form "$(\mathsf{RHandle}\langle r\rangle\ h)\ \{e\}$" creates a local region (i.e., a region of kind LocalRegion) using the default allocation policy VT. Figure 6-18 presents the translation for "$(\mathsf{RHandle}\langle r\rangle\ h)\ \{e\}$". The code from Figure 6-18 works as follows:

1. Create region h = new VTLocalRegion(false)
2. Declare a class RE that implements the Runnable interface. Its run() method serve as a wrapper for the expression e. We introduce several fields in RE for dealing with the free variables of e. For each such variable $v \neq$ this, RE has a field v of appropriate type. If this appears in e, we create a field with a fresh name _this (this refers to another object in the methods of RE). The body of the run() method consists of e, with each free occurrence of this substituted by _this.
3. We create an instance *re* of RE in the current region and initialize its fields with the free variables of e.
4. Execute e: h.enter(*re*);
5. Retrieve the (possibly changed) values of the free variables of e from *re*'s fields.

6.B.3 Creating a Shared Region

The translation for "$(\mathsf{RHandle}\langle \mathit{rkind}:\mathit{rpol}\ r\rangle\ h)\ \{e\}$" is similar to that for a local region (Figure 6-18), with two important differences.

First, we change Line 1 as follows:

```
1': Irkind h =
      if(rpol = LT(size)) new LTrkind(true, size);
      else new VTrkind(true);
```

Accordingly, field h of class RE (Line 2) has now type $\mathsf{I}\mathit{rkind}$.

Second, we change the body of the run() method of class RE (Line 3 in Figure 6-18) as follows:

```
public void run() {
  h.enterRegion();
  try {
    translation of e[id/this]
  } finally {
     h.exitRegion();
  }
}
```

```
import javax.realtime.*;
import java.util.concurrent.atomic.*;

public class LTrkind extends LTMemory implements Irkind {
  public  rkindSubs getSubs() { return  subs; }
  private rkindSubs subs;

  public  rkindFields getFields() { return (rkindFields) getPortal(); }

  public  boolean isFlushed() { return isFlushed; }
  private boolean isFlushed;

  public  boolean isASubregion() { return isASubregion; }
  public  void    setIsASubregion(boolean value) { isASubregion = value; }
  private boolean isASubregion;

  private AtomicInteger n_inside  = new AtomicInteger(0);   // number of threads inside
  private AtomicInteger getNInside()  { return n_inside; }
  private AtomicInteger n_exiting = new AtomicInteger(0);   // number of threads exiting
  private AtomicInteger getNExiting() { return n_exiting; } // See JSR 166

  public LTrkind(boolean isASubregion, int size) {
    super(size);
    this.isFlushed = true;
    this.isASubregion = isASubregion;
    this.subs = new rkindSubs();
    ∀ subregion (srkind_s : rpol_s rsub) ∈ rkind, generate
      subs.rsub =
        if rpol_s = LT(size)  new LTsrkind_s(true,size);
        else new VTsrkind_s(true);
  }

  public void enterRegion() { ... presented later ... }
  public void exitRegion()  { ... presented later ... }

  private boolean extraFlushingTest() {
    if(!isASubregion()) return true;
    rkindFields fields = this.getFields();
    if(fields != null) {
      ∀ field (t fd)∈rkind, generate
        if(fields.fd != null) return false;
    }
    rkindSubs subs = this.getSubs();
    ∀ subregion (srkind_s : rpol_s rsub) ∈ rkind, generate
      if(!subs.rsub.isFlushed()) return false;
    return true;
  }

  private void flush() {
    isFlushed = true;  setPortal(null);
    rkindSubs subs = getSubs();
    if(!isASubregion()) {
      ∀ subregion (rkind_s : rpol_s rsub) ∈ rkind, generate
        IRegion sr = subs.rsub;  if(!sr.isFlushed()) sr.flush();
}}}
```

Figure 6-17: Declaration of Class LT*rkind*

```
{
   // create a new RTSJ region
1: ILocalRegion h = new VTLocalRegion(false);

   // create a Runnable object to wrap the code of e
   static class RE implements Runnable {
     // one field for each free variable of e
     if h ∈ FreeVars(e)
2:     public ILocalRegion h;
     ∀ v ∈ FreeVars(e) \ {h, this}; let t be its type in the environment
       public t v;
     if this ∈ FreeVars(e); let t be its type in the environment
       public t _this;

     public void run() {
3:     translation of e[_this/this]
     }
   }
   RE re = new RE();

   // store h and all free variables of e in re's fields
   re.h = h;
   ∀ v ∈ FreeVars(e) \ {h, this}
     re.v = v;
   if this ∈ FreeVars(e)
     re._this = this;

   // evaluate e
   ((MemoryArea) h).enter(re);

   // restore the values of e's free variables
   ∀ v ∈ FreeVars(e) \ {h, this}
     v = re.v;
}
```

where RE, re, and _this are fresh identifiers.

Figure 6-18: Translation for "(RHandle$\langle r \rangle$ h) $\{e\}$"

```
public class LTrkind ... {

  ... as before ...

  public void enterRegion() {
    int x = getNInside().add(1);
    // Wait until exiting threads finish exiting
    while (getNExiting().get() > 0) sleep(1000);
    // 1st entering thread creates portal
    if (x == 1) {
      isFlushed = false;
      if (getPortal() == null)
        setPortal(new rkindFields());
    } else {
      // Others wait until portal is created
      while (getPortal() == null) sleep(1000);
    }
  }

  public void exitRegion() {
    // Begin exiting the region
    getNExiting().add(1);
    int x = getNInside().add(-1);
    if ((x == 0) && extraFlushingTest())
      flush();
    // Finish exiting the region
    h.getNExiting().add(-1);
  }
}
```

Figure 6-19: Declaration of Class LT*rkind* (Part II): enterRegion() and exitRegion()

At the beginning of the run() method, we call the method enterRegion() to do some bookkeeping for region r. Next, we execute the expression e in a try-finally block. This way, we ensure that no matter how e terminates, we call the cleanup method exitRegion() for region r right before run() terminates.

Figure 6-19 completes the definition of class LT*rkind* from Figure 6-17 by providing the implementation of enterRegion() and exitRegion(). Most of these methods' code deals with synchronization issues. The code is more complicated than what is required for the case of a top-level shared region: e.g., as the thread that creates the region is also the one to enter it first, enterRegion() could have been simplified significantly. For space reason, we present the full versions of these methods (that are valid even when an LT*rkind* region is used as a subregion in future sections) instead of going through several specialized versions. We explain these methods in the most general context: at any moment, several threads may simultaneously attempt to enter/exit the region.

The last thread to use a region (possibly a subregion) *must* call its flush() method if extraFlushingTest() is satisfied: otherwise, the portal object of the corresponding RTSJ memory area remains non-null, and the memory area is never flushed. When a thread enters the region, it has to create its portal object if it does not exists yet (e.g., if the (sub)region has been flushed and not re-entered yet). Therefore, both enterRegion() and exitRegion() may write the portal field of the corresponding memory area.

If there is no synchronization between enterRegion() and exitRegion(), then the following scenario is possible:

1. Thread T1 starts exiting the region; it checks the flushing conditions and decides to flush; however, it is pre-empted by the scheduler before doing the actual flushing;
2. Thread T2 enters the region;
3. Thread T1 flushes the region; in particular, it sets the portal of the underyling memory area to null;
4. Thread T2 attemps to use a portal field and raises a NullPointerException.

We avoid this race condition by a tricky synchronization algorithm that uses atomic operations defined in JSR 166 [95]. Our synchronization ensures safety: when a thread is using a region, no other thread can flush the same region; and when a thread is using a region, the region has a non-null portal. Because the set of subregions that the normal threads use is disjoint from the set of threads that the realtime threads use, our synchronization does not create priority inversion problems (see discussion at the end of Section!6.3). Our algorithm maintains two AtomicInteger[2] counters:

- n_inside, that is returned by getNInside() and maintains the count of the threads that currently use the region;
- n_exiting, that is returned by getNExiting() and maintains the count of the threads that started the exitRegion() method but did not complete it.

The method enterRegion() starts by incrementing the number of threads using the region. Until this thread exits the region, no exiting thread can decide to flush the region if it has not done so yet. The first while loop from enterRegion() ensures that no thread enters the region until all the exiting threads finished exiting it. If the region does not have a portal object, the thread that changed the n_inside counter from 0 to 1 is responsible with creating that object; the other threads attempting to enter the region have to wait for the portal object to be ready (due to the second while loop).

The "useful" part of exitRegion() decrements n_inside and calls flush() if all flushing conditions are met. It is protected from interferences with enterRegion() by the counter n_exiting, that is appropriately maintained at the beginning and at the end of exitRegion().

Note that the RTSJ platform already maintains the count of the threads executing inside a region. This counter is almost identical to our n_inside. However, there is a notable difference: the RTSJ counter is not atomic; e.g., if we try to use it, the test "x == 1" from enterRegion() may fail if two threads enter the same subregion simultaneously (the value returned by getReferenceCount() may jump from 0 to 2).

[2]For the purpose of this report, an AtomicInteger is an integer such that we can *atomically* increment/decrement it and read its new value.

6.B.4 Entering a Subregion

The translation for "(RHandle$\langle r\rangle$ $h = h_2.rsub$) $\{e\}$" is very similar to the one from Section 6.B.3. The only difference is that now, instead of creating a region, we simply read one and use it. The beginning of the translation (Line 1') becomes:

1'': I*srkind* h = h_2.getSubs().*rsub*;

where *rkind* is the kind of the subregion *rsub*.

6.B.5 Creating a Subregion

The translation for "(RHandle$\langle r\rangle$ $h =$ new $h_2.rsub$) $\{e\}$" is very similar to the one from Section6.B.3. Only the beginning of the translation changes as follows:

```
1''': static class RE2 implements Runnable {
        Isrkind h;
        Isrkind2 h2;
        public void run() {
          h =
            if rpol_s = LT(size) new LTrkind_s(true,size);
            else new VTrkind_s(true);
          h2.getSubs().rsub = h;
        }
      }
      MemoryArea ma = MemoryArea.getMemoryArea(h2);
      h2.getSubs().rsub.setIsASubregion(false);
      RE2 re2 = new RE2();
      re2.h2 = h2;
      ma.enter(re2);
      Isrkind h = re.h;
```

where $rkind_s$ is the kind of the subregion *rsub*, $rpol_s$ is its allocation policy, and *RE*, *re*, and *ma* are fresh identifiers. The above code works as follows:

1. To be consistent with our representation of regions (see Section 6.B.1), we allocate the memory area objects for the new subregion (and its subregions) in the same memory area where the previous subregion was allocated. Most of the above code deals with technical details related to this operation: by wrapping the creation of the new subregion in a Runnable object, we ensure that all objects used for the representation of that subregion are allocated in the appropriate region.

2. We detatch the subregion from its parent: "h_2.subs.$rsub$.setIsASubregion(false)" to record the fact that it cannot be entered from its parent. The first time its counter becomes zero, it will be flushed (and subsequently deleted along with its subregions).

6.B.6 Manipulating Region Fields

We translate "$h.fd$" as follows:

((*rkind*Fields) h.getPortal()).*fd*

where *rkind* is the kind of the region r that h is a handle of, i.e., in the type environment, h has type RHandle$\langle r\rangle$.

```
{
  static class T extends javax.realtime.RealtimeThread {
    ∀ i ∈ {0,...,m}, generate one field to store the value of v_i (of type t_i):
      t_i vi;
    public void run() {
      for(int i = 0; i < getMemoryAreaStackDepth(); i++) {
        IRegion isr = (IRegion) getOuterMemoryArea(i);
        isr.enterRegion();
      }
      try {
        v0.mn(v1,..., vm);
      } finally {
        for(int i = 0; i < getMemoryAreaStackDepth(); i++) {
          IRegion isr = (IRegion) getOuterMemoryArea(i);
          isr.exitRegion();
        }
      }
    } // end of run()
  }
  T t = new T();
  ∀ i ∈ {0,...,m}, generate one line of the form:
    t.vi = v_i;
  t.start();
}

where T and t are fresh identifiers.
```

Figure 6-20: Translation for "fork $v_0.mn\langle o_{1..n}\rangle(v_{1..m})$"

The translation for "$h.fd = v$" is similar:

```
((rkindFields) h.getPortal()).fd = v
```

6.B.7 Allocating an Object

There are no constructors in the language we presented so far. However, they are trivial to add: an expression of the form "new $cn\langle o_{1..n}\rangle(e_{1..m})$" desugars into a "new $cn\langle o_{1..n}\rangle$" followed by a call to the appropriate constructor. We translate "new $cn\langle o_{1..n}\rangle(e_{1..m})$" as follows:

1. First, we generate Java code to retrieve the memory area where the new object is allocated: the region where o_1 is allocated (if o_1 is an object) or the region o_1 stands for (if o_1 is a region).
2. Next, we generate a call to newInstance, to allocate a new object in the memory area that the code generated at 1 evaluates to.
3. We recursively translate e_1, ..., e_m (the arguments of the constructor).
4. Finally, we generate a call to the appropriate constructor.

The only non-trivial step is the first one: retrieving the memory area where we allocate the new object. The type rule for new already checked that $E \vdash_{\text{av}} \mathsf{RH}(o_1)$, i.e., a handle for this region is available at runtime, even after type-erasure (see Section 6.4). We use the typechecker judgments to retrieve that region. Notice that each of the rules that prove a statement of the form $E \vdash_{\text{av}} \mathsf{RH}(o)$ has at most one such statement among its preconditions. Therefore, if we consider the part of the proof tree for $E \vdash_{\text{av}} \mathsf{RH}(o_1)$ that corresponds only to this kind of rules, we obtain a chain. This "reasoning" chain starts with either [AV_THIS] or [AV_HANDLE]. In the case of [AV_THIS] we generate the call "MemoryArea.getMemoryArea(this)".

In the second case, i.e., [AV_HANDLE], the typing environment E contains a handle for the appropriate region. Let r and h be the region and the region handle from the specific instantiation of the rule [AV_HANDLE]. If r is the special region heap, we generate a call to HeapMemory.instance(), an RTSJ method that retrieves the (unique) heap memory area. Similarly, if r is the special region immortal, we generate a call to ImmortalMemory.instance(). Otherwise, in the translated code, h is a local variable that points to the region we allocate in; we directly generate the code "(MemoryArea) h".

Optimization: newInstance allows us to allocate an object in any region. However, it currently uses reflection, e.g., for passing the class of the allocated object. Hence, it is less efficient than new, that allocates only in the current region. The typechecker knows the current region r_{cr} for the new expression that we translate. For [AV HEAP], [AV IMMORTAL] and [AV HANDLE], if r_{cr} is identical to the region we allocate in, we use new. We can apply this optimization even in the case of [AV THIS], if the typechecker can prove that $r_{cr} \succeq$ this.

6.B.8 Forking a Thread

Figure 6-20 presents the translation for an expression of the form "fork $v_0.mn\langle o_{1..n}\rangle(v_{1..m})$". The resulting code works as follows:

1. In Java, threads are objects whose class is a subclass of java.lang.Thread. Programmers create thread objects using new and start them by invoking their start() method. start() starts a thread whose body is the run() method of the thread object. In RTSJ, threads that want to use regions have to subclass javax.realtime.RealtimeThread, which itself subclasses java.lang.Thread. Accordingly, we define a class T for our thread. T has one field vi to store the value of each variable v_i.

2. The run() method of T invokes mn with the right receiver and parameters. When the thread terminates, each region that is still on its stack of regions is exited. Therefore, for each such region, if our conditions for flushing it are fulfilled, we need to make sure that it meets the conditions for being flushed by RTSJ. Fortunately, RTSJ offers methods that allow us to examine the stack of memory areas associated with a thread.

3. We create a instance of class T, generate code to store the result of each variable v_i in the appropriate field and next start the thread.

The only difference in the translation for "RT_fork $v_0.mn\langle o_{1..n}\rangle(v_{1..m})$" is that we subclass T from NoHeapRealtimeThread, instead of RealtimeThread. In RTSJ, a NoHeapRealtimeThread is not interrupted by the garbage collector because it cannot manipulate heap references. RTSJ ensures this using dynamic checks. SafeJava ensures this statically, so we can remove these dynamic checks if the RTSJ platform allows us to do so.

Chapter 7

Conclusions

Making software reliable is one of the most important technological challenges facing our society today. This thesis presents a new type system that addresses this problem by statically preventing several important classes of programming errors. If a program type checks, we guarantee at compile time that the program does not contain any of those errors.

We designed our type system in the context of a Java-like object-oriented language; we call the resulting system *SafeJava*. The SafeJava type system offers significant software engineering benefits. Specifically, it provides a statically enforceable way of specifying object encapsulation and enables local reasoning about program correctness; it combines effects clauses with encapsulation to enable modular checking of methods in the presence of subtyping; it statically prevents data races and deadlocks in multithreaded programs, which are known to be some of the most difficult programming errors to detect, reproduce, and eliminate; it enables software upgrades in persistent object stores to be defined modularly and implemented efficiently; it statically ensures memory safety in programs that manage their own memory using regions; and it also statically ensures that real-time threads in real-time programs are not interrupted for unbounded amounts of time because of garbage collection pauses. Moreover, SafeJava provides all the above benefits in a common unified type system framework, indicating that seemingly different problems such as encapsulation, synchronization issues, software upgrades, and memory management have much in common.

We have implemented several Java programs in SafeJava. Our experience shows that SafeJava is expressive enough to support common programming patterns, its type checking is fast and scalable, and it requires little programming overhead. In addition, the type declarations in SafeJava programs serve as documentation that lives with the code, and is checked throughout the evolution of code. The SafeJava type system thus has significant software engineering benefits and it offers a promising approach for improving software reliability.

Bibliography

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data*, May 1995.

[2] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, January 2004.

[3] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[4] Alex Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation (PLDI)*, June 1995.

[5] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronizations from Java programs. In *Static Analysis Symposium (SAS)*, September 1999.

[6] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[7] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

[8] Malcolm P. Atkinson, Mikhail A. Dmitriev, Craig Hamilton, and Tony Printezis. Scalable and recoverable implementation of object evolution for the PJama 1 platform. In *Persistent Object Systems (POS)*, September 2000.

[9] Malcolm P. Atkinson, Mick J. Jordan, Laurent Daynes, and Susan Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Persistent Object Systems (POS)*, May 1996.

[10] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Principles of Programming Languages (POPL)*, January 2003.

[11] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[12] Henry G. Baker. List processing in real-time on a serial computer. In *Communications of the ACM (CACM) 21(4)*, April 1978.

[13] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD International Conference on Management of Data*, May 1987.

[14] Elisa Bertino, Giovanna Guerrini, and Luca Rusca. Object evolution in object databases. In *B. Franhoefer and R. Pareschi, editors, Dynamic Worlds, Kluwer Academic Publishers*, 1999.

[15] Bruno Blanchet. Escape analysis for object-oriented languages. Application to Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[16] Jeff Bogda and Urs Holzle. Removing unnecessary synchronization in Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[17] Boris Bokowski and Jan Vitek. Confined types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1999.

[18] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000. Latest version available from http://www.rtj.org.

[19] Chandrasekhar Boyapati. Towards an extensible virtual machine. Technical Report TR-842, MIT Laboratory for Computer Science, April 2002.

[20] Chandrasekhar Boyapati. JPS: A distributed persistent Java system. SM thesis, Massachusetts Institute of Technology, September 1998.

[21] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[22] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (ECOOP IWACO)*, July 2003.

[23] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[24] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

[25] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[26] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[27] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[28] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and sharing. In *European Conference for Object-Oriented Programming (ECOOP)*, June 2001.

[29] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[30] Robert Bretl et al. The GemStone data management system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. 1989.

[31] Per Brinch-Hansen. The programming language Concurrent Pascal. In *IEEE Transactions on Software Engineering SE-1(2)*, June 1975.

[32] Robert Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[33] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Symposium on Operating System Principles (SOSP)*, October 1997.

[34] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Principles of Programming Languages (POPL)*, January 2002.

[35] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998.

[36] Yin Cheung. Lazy schema evolution in object-oriented databases. SM thesis, Massachusetts Institute of Technology, September 2001.

[37] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[38] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[39] Morten V. Christiansen, Fritz Henglein, Henning Niss, and Per Velschow. Safe region-based memory management for objects. Technical Report D-397, DIKU, University of Copenhagen, October 1998.

[40] Stewart M. Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, June 1992.

[41] David G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[42] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, June 2001.

[43] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[44] David G. Clarke and Tobias Wrigstad. External uniqueness. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2003.

[45] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[46] Angelo Corsaro and Douglas Schmidt. The design and performance of the jRate Real-Time Java implementation. In *International Symposium on Distributed Objects and Applications (DOA)*, October 2002.

[47] Angelo Corsaro and Douglas Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, September 2002.

[48] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, January 1999.

[49] Viviane M. Crestana-Jensen, Amy J. Lee, and Elke A. Rundensteiner. Consistent schema version removal: An optimization technique for object-oriented views. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 12(2)*, March 2000.

[50] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

[51] Christine Delcourt and Roberto Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1991.

[52] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[53] Robert DeLine and Manuel Fahndrich. Adoption and focus: Practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[54] Morgan Deters and Ron Cytron. Automated discovery of scoped memory regions for Real-Time Java. In *International Symposium on Memory Management (ISMM)*, June 2002.

[55] Morgan Deters, Nicholas Leidenfrost, and Ron Cytron. Translation of Java to Real-Time Java using aspects. In *International Workshop on Aspect-Oriented Programming and Separation of Concerns*, August 2001.

[56] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Compaq Systems Research Center, July 1998.

[57] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[58] O. Deux et al. The story of O2. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 2(1)*, March 1990.

[59] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD)*, May 1991.

[60] Mikhail A. Dmitriev. Safe class and data evolution in large and long-lived Java applications. Technical Report TR-2001-98, Sun Microsystems, August 2001.

[61] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *ECOOP*, 2001.

[62] Dominic Duggan. Type-based hot swapping of running modules. Technical Report SIT CS 2001-7, Stevens Institute of Technology, Hoboken, NJ 07030, October 2001.

[63] Dawson R. Engler, David Yu Chen, Seth Hallem, Andy Chon, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, October 2001.

[64] Fabrizio Ferrandina and Guy Ferran. Schema and database evolution in the O2 object database system. In *Very Large Data Bases (VLDB)*, 1995.

[65] Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Measuring the performance of immediate and deferred updates in object database systems. In *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, 1995.

[66] Cormac Flanagan and Martin Abadi. Object types against races. In *Conference on Concurrent Theory (CONCUR)*, August 1999.

[67] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming (ESOP)*, March 1999.

[68] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[69] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[70] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[72] David Gay and Alex Aiken. Memory management with explicit regions. In *Programming Language Design and Implementation (PLDI)*, June 1998.

[73] David Gay and Alex Aiken. Language support for regions. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[74] Ovidiu Gheorghioiu. Statically determining memory consumption of real-time Java threads. MEng thesis, Massachusetts Institute of Technology, June 2002.

[75] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Principles of Programming Languages (POPL)*, January 2003.

[76] Jean-Yves Girard. Linear logic. In *Theoretical Computer Science*, 1987.

[77] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[78] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1999.

[79] Dan Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, January 2003.

[80] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[81] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[82] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in computer Science. Springer Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andres Modet, and Jeannette M. Wing.

[83] Michael Hicks, Jonathan Moore, and Scott Nettles. Dynamic software updating. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[84] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Unique pointers and reference counting in Cyclone, October 2003. Unpublished Manuscript.

[85] Gilsi Hjalmtysson and Rober Gray. Dynamic C++ classes—A lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, June 1998.

[86] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1991.

[87] John Hogg, Doug Lea, Alan Wills, and Dennis de Champeaux. The Geneva convention on the treatment of object aliasing. In *OOPS Messenger 3(2)*, April 1992.

[88] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. In *Principles of Programming Languages (POPL)*, January 2001.

[89] JavaSoft. Inner class specification, February 1997. Available at http://java.sun.com-/products/JDK/1.1.

[90] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[91] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Principles of Programming Languages (POPL)*, January 1993.

[92] Joseph A. Korty. Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *USENIX Winter Technical Conference*, January 1989.

[93] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Principles of Programming Languages (POPL)*, January 2002.

[94] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language Euclid. In *Sigplan Notices, 12(2)*, February 1977.

[95] Doug Lea. JSR166: Concurrency utilities.

[96] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, May 1996.

[97] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[98] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[99] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[100] Barbara S. Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1990.

[101] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[102] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1999.

[103] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[104] Barbara Liskov, Alan Snyder, Russell R. Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. In *Communications of the ACM (CACM) 20(8)*, August 1977.

[105] Andrew Lister. The problem of nested monitor calls. In *Operating Systems Review 11(3)*, July 1977.

[106] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[107] Ole L. Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[108] Naftaly Minsky. Towards alias-free pointers. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1996.

[109] Anders Moeller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[110] Carroll Morgan. The specification statement. *Transactions on Programming Languages and Systems (TOPLAS) 10(3)*, July 1998.

[111] Peter Muller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming.* 1999.

[112] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[113] James Noble. Iterators and encapsulation. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, June 2000.

[114] Object Design Inc. *ObjectStore Advanced C++ API User Guide Release 5.1*, 1997.

[115] Objectivity Inc. *Objectivity Technical Overview, Version 6.0*, 2001.

[116] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1987.

[117] William Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[118] Martin Rinard. Analysis of multi-threaded programs. In *Static Analysis Symposium (SAS)*, July 2001.

[119] Erik Ruf. Effective synchronization removal for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[120] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems (TOPLAS) 20(1)*, January 1998.

[121] Alexandru Salcianu. Pointer analysis and its applications for Java programs. MEng thesis, Massachusetts Institute of Technology, September 2001.

[122] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Principles and Practice of Parallel Programming (PPoPP)*, June 2001.

[123] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Symposium on Operating Systems Principles (SOSP)*, October 1997.

[124] The Sustainable Computing Consortium. http://www.sustainablecomputing.org.

[125] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1986.

[126] J. Mike Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 2nd edition, 1992.

[127] Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, January 1993.

[128] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. In *Information and Computation 132(2)*, February 1997.

[129] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (POPL)*, January 1994.

[130] Versant Object Technology. *Versant User Manual*, 1992.

[131] Mirko Viroli. Parametric polymorphism in Java: An efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, March 2001.

[132] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[133] Christoph von Praun and Thomas Gross. Object-race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[134] Philip Wadler. Linear types can change the world. In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. 1990.

[135] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[136] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, June 1995.

[137] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. In *Information and Computation 115(1)*, November 1994.

[138] Roberto Zicari. A framework for schema updates in an object-oriented database systems. In *International Conference on Data Engineering (ICDE)*, April 1991.

Chandrasekhar Boyapati

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, NE43-631, Cambridge, MA 02139
(617) 253-6031

chandra@lcs.mit.edu
http://cag.lcs.mit.edu/~chandra

Education

09/98 - 12/03 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Doctor of Philosophy in Electrical Engineering and Computer Science, February 2004
Thesis: *SafeJava: A Unified Type System for Safe Programming*
Martin Rinard, advisor

Minor in Approximation Algorithms

09/96 - 08/98 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Master of Science in Electrical Engineering and Computer Science, September 1998
Thesis: *JPS: A Distributed Persistent Java System*
Barbara Liskov, advisor

08/92 - 05/96 **Indian Institute of Technology (IIT)** **Madras, India**
Bachelor of Technology in Computer Science and Engineering, July 1996
Thesis: *Worst Case Efficient Data Structures for Priority Queues and Deques with Heap Order*
C. Pandu Rangan, advisor

Research Interests

My primary research interest is software reliability, which spans the spectrum from programming languages, through program analysis, to software engineering. I am interested in all approaches for helping programmers write and maintain reliable software. I am particularly interested in developing programming languages with strong type systems to improve reliability.

Research Experience

09/00 - 12/03 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Graduate Research Assistant in the Programming Analysis and Compilation Group
Martin Rinard, advisor

09/96 - 05/00 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Graduate Research Assistant in the Programming Methodology Group
Barbara Liskov, advisor

06/00 - 08/00 **Microsoft Research (MSR)** **Redmond, WA**
Research Intern in the Advanced Programming Languages Group
Erik Ruf, mentor

06/99 - 08/99 **Xerox Palo Alto Research Center (PARC)** **Palo Alto, CA**
Research Intern in the Aspect-Oriented Programming Group
Gregor Kiczales, mentor

01/95 - 05/96 **Indian Institute of Technology (IIT)** **Madras, India**
Student Researcher in the Theory Laboratory
C. Pandu Rangan, advisor

Teaching Experience

01/99 - 05/99 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Teaching Assistant for Computer Systems Engineering (6.033)
Frans Kaashoek, lecturer

01/98 - 05/98 **Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Teaching Assistant for Laboratory in Software Engineering (6.170)
Barbara Liskov and Daniel Jackson, lecturers

08/95 - 11/95 **Indian Institute of Technology (IIT)** **Madras, India**
Teaching Assistant for Introduction to Computing (CS 110)
C. Pandu Rangan, lecturer

Conference Publications

2003 [1] **Lazy Modular Upgrades in Persistent Object Stores**
Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, Steven Richman.
ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2003.

2003 [2] **Ownership Types for Safe Region-Based Memory Management in Real-Time Java**
Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Martin Rinard.
ACM Conference on Programming Language Design and Implementation (PLDI), June 2003.

2003 [3] **Ownership Types for Object Encapsulation**
Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira.
ACM Symposium on Principles of Programming Languages (POPL), January 2003.
This is an invited paper.

2002 [4] **Ownership Types for Safe Programming: Preventing Data Races and Deadlocks**
Chandrasekhar Boyapati, Robert Lee, Martin Rinard.
ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), November 2002.

2002 [5] **Korat: Automated Testing Based on Java Predicates**
Chandrasekhar Boyapati, Sarfraz Khurshid, Darko Marinov.
ACM International Symposium on Software Testing and Analysis (ISSTA), July 2002.
This paper won an **ACM SIGSOFT distinguished paper award.**

2001 [6] **A Parameterized Type System for Race-Free Java Programs**
Chandrasekhar Boyapati and Martin Rinard.
ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2001.

Workshop Publications

2002 [7] **Safe Runtime Downcasts With Ownership Types**
Chandrasekhar Boyapati, Robert Lee, Martin Rinard.
ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO), July 2003.

2002 [8] **Safe Concurrent Programming in Java**
Chandrasekhar Boyapati, Robert Lee, Martin Rinard.
MIT LCS/AI Student Oxygen Workshop (SOW), July 2002.

2001 [9] **A Type System for Preventing Data Races**
Chandrasekhar Boyapati and Martin Rinard.
MIT LCS/AI Student Oxygen Workshop (SOW), July 2001.

Technical Reports

2002 [10] **Towards an Extensible Virtual Machine**
Chandrasekhar Boyapati.
Technical Report MIT-LCS-TR-842, Laboratory for Computer Science, MIT, April 2002.

1995 [11] **Relaxed Fibonacci Heaps: An Alternative to Fibonacci Heaps With Worst Case Rather Than Amortized Time Bounds**
Chandrasekhar Boyapati and C. Pandu Rangan.
Technical Report TR-TCS-95-07, Theoretical Computer Science Laboratory, IIT Madras, November 1995.

1995 [12] **On O(1) Concatenation of Deques with Heap Order**
Chandrasekhar Boyapati and C. Pandu Rangan.
Technical Report TR-TCS-95-05, Theoretical Computer Science Laboratory, IIT Madras, March 1995.

Theses

2004 [13] **SafeJava: A Unified Type System for Safe Programming**
Doctor of Philosophy, Electrical Engineering and Computer Science, MIT, February 2004.

1998 [14] **JPS: A Distributed Persistent Java System**
Master of Science, Electrical Engineering and Computer Science, MIT, September 1998.

1996 [15] **Worst Case Efficient Data Structures for Priority Queues and Deques with Heap Order**
Bachelor of Technology, Computer Science and Engineering, IIT Madras, India, May 1996.

Patents

2002 [16] **Aspect-Oriented Programming**
Gregor Kiczales, John Lamping, Cristina Lopes, James Hugunin, Erik Hilsdale, Chandrasekhar Boyapati.
U.S. Patent No. 6,467,086, issued October 2002.

Program Committees

2004	ACM Conference on Programming Language Design and Implementation (PLDI)
2002	MIT LCS/AI Student Oxygen Workshop (SOW)

Refereeing

2004	ACM Symposium on Principles of Programming Languages (POPL)
2003	Journal of Functional Programming (JFP)
2003	ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)
2003	International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)
2003	ACM Symposium on Principles of Programming Languages (POPL)
2002	ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)
2002	International Static Analysis Symposium (SAS)
2002	MIT LCS/AI Student Oxygen Workshop (SOW)
2002	European Conference on Object-Oriented Programming (ECOOP)

Advising

2002 **M.Eng. Thesis, Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Along with Martin Rinard, informally co-supervised Robert Lee, a student whose M.Eng. thesis on *Extending the Java Language for the Prevention of Data Races* is related to my Ph.D. thesis

1997 **RSI Summer Research Project, Massachusetts Institute of Technology (MIT)** **Cambridge, MA**
Supervised Athinodoros Panagiotidis, a research intern in the Research Science Institute (RSI) summer program, jointly sponsored by Center for Excellence and Education (CEE) and MIT

Invited Talks

04/2003 **Microsoft Research** **Redmond, WA**
Jim Larus, host

04/2003 **University of Massachusetts at Amherst** **Amherst, MA**
Lori Clarke, host

04/2003 **Rice University** **Houston, TX**
Keith Cooper, host

04/2003 **University of Chicago** **Chicago, IL**
David MacQueen, host

04/2003 **University of California at San Diego** **San Diego, CA**
Brad Calder, host

03/2003 **IBM T. J. Watson Research Center** **Yorktown Heights, NY**
David Bacon, host

03/2003 **University of Michigan at Ann Arbor** **Ann Arbor, MI**
Brian Noble, host

11/2002 **Microsoft Research** **Redmond, WA**
Jim Larus, host

02/2002 **New England Programming Languages and Systems Symposium (NEPLS)** **Cambridge, MA**
Michael Ernst, chair

10/2001 **Church Seminar, Boston University** **Boston, MA**
Assaf Kfoury, host

Selected Honors

2002	ACM SIGSOFT distinguished paper award [5].
1996	Gordon Wu Fellowship, Princeton's most prestigious award for graduate study in engineering. Declined.
1992	Ranked 11^{th} out of over 100,000 students in the All India Joint Entrance Examination for admission to IITs.
1992	Ranked among the top 20 out of over 400,000 students in the West Bengal Higher Secondary Examination.
1992	Recipient of National Talent Search Scholarship awarded to top 0.5% of over 150,000 students, based on a nationwide test conducted by the National Council of Educational Research and Training (NCERT), India.
1988 & 1989	Got the top rank among nearly 20,000 students in the Science Aptitude and Talent Search Test conducted by All India Science Teachers Association, West Bengal branch, both in 1988 and 1989.