Checking the Data Sharing Strategies of Concurrent Systems Level Code

Zachary Ryan Anderson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-59
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-59.html

May 11, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Checking the Data Sharing Strategies of Concurrent Systems Level Code

by

Zachary Ryan Anderson

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric A. Brewer, Chair
Professor George C. Necula
Professor Rastislav Bodik
Professor Theodore A. Slaman

Spring 2010

The dissertation of Zachary Ryan Anderson, titled Checking the Data Sharing Strategies of Concurrent Systems Level Code, is approved:

Chair ____ Date ________________

____ Date ________________

____ Date ________________

____ Date ________________

University of California, Berkeley

Checking the Data Sharing Strategies of Concurrent Systems Level Code

Copyright 2010
by
Zachary Ryan Anderson

Abstract

Checking the Data Sharing Strategies of Concurrent Systems Level Code

by

Zachary Ryan Anderson

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric A. Brewer, Chair

Due to the high degree of control and performance that it affords, programmers use the C language for writing low-level systems software. That it is difficult to write programs in C is not a problem in and of itself. Modern languages deal with its safety issues through innovations in language design, programming interfaces and runtime environments. However, due to the amount and complexity of existing C code, rewriting all of it in modern languages is likely infeasible, and the absence of a modern replacement for C having all of its advantages ensures that it will continue to be used to create new software for the foreseeable future.

The goal of the Ivy compiler is to provide an evolutionary pathway from C to a language with stronger safety guarantees. Because rewriting software all at once is not an option, the design philosophy of Ivy is to provide ways for the programmer to transition software in a modular fashion from C to a language with the desired safety guarantees. Given these requirements, Ivy provides memory- and type-safety to sequential programs with two components, one called Deputy, and the other called Heapsafe. However, Deputy and Heapsafe are unsound when faced with multi-threaded programs.

This dissertation presents SharC, an extension to Ivy that provides for safe concurrent programming, and Shelters, a deadlock-free, pessimistic implementation of atomic sections that avoids software transactional memory and whole-program analysis. SharC allows programmers to declare how objects in multi-threaded programs are shared among threads. SharC uses a combination of static and dynamic analysis to enforce these "sharing modes." Additionally, since objects in programs can go through several phases, SharC allows programmers to declare where the sharing mode of an object changes. SharC checks these sharing mode changes by requiring that there is only one reference to an object when its sharing mode changes. Further, SharC provides features for applying and changing these sharing modes across complex data structures. Finally, our shelter-based atomic sections are presented as a sharing mode of SharC that can provide the convenience of atomic sections while achieving performance comparable with explicit locking.

We evaluate our implementation of SharC and Shelters on over 1.5 million lines of application and benchmark code, and observe manageable overheads both in terms of performance and programmer effort.

Contents

List of Figures v

List of Tables vii

I Introduction 1

1 Introduction 3
1.1 Unintended Sharing . . . 3
1.2 Deadlock . . . 4
1.3 The C Programming Language . . . 5
1.4 The Ivy Compiler . . . 6
1.5 SharC . . . 7
1.6 Shelters . . . 8
1.7 Evaluation . . . 9
1.8 Summary . . . 10

2 Ivy 13
2.1 Deputy . . . 13
2.2 Heapsafe . . . 17
2.3 SharC and Shelters . . . 18
2.4 Putting It All Together . . . 18

3 Related Work 21
3.1 Static Race Detection and Model Checking . . . 22
3.2 Dynamic Race Detection . . . 23
3.3 Ownership and Region Type Systems . . . 24
3.4 Atomic Sections . . . 25

II SharC 27

4 Design of SharC 29
4.1 Design of SharC . . . 29

4.1.1 An Example . . . 31
4.2 Limitations of the Single-Object Sharing Cast . . . 34
4.3 Groups . . . 36
4.3.1 Group Annotations and Casts . . . 37
4.3.2 The Group Leader . . . 38
4.3.3 Operations on Groups . . . 39
4.3.4 Instrumentation . . . 39
4.3.5 A More Complex Example . . . 41
4.3.6 Limitations of Groups Design . . . 43

5 Formalizing SharC **45**
5.1 The Soundness of SharC with Groups . . . 45
5.1.1 Static Semantics . . . 48
5.1.2 Operational Semantics . . . 51
5.1.3 Soundness . . . 54

6 Implementing SharC **55**
6.1 Thread-Escape Analysis . . . 55
6.2 Runtime Checks . . . 57
6.2.1 Tracking Reader and Writer Sets . . . 57
6.2.2 Tracking Held Locks . . . 57
6.2.3 Checking Sharing Casts . . . 57
6.2.4 Maintaining Reference Counts . . . 58
6.3 Groups . . . 59
6.4 The C Library . . . 59
6.5 Failure Modes . . . 60
6.6 Limitations . . . 60

III Shelters **61**

7 Design of Shelters **63**
7.1 Design of Shelters . . . 63
7.1.1 Example . . . 64
7.1.2 Shelter Hierarchy . . . 67
7.1.3 Condition Variables . . . 68
7.1.4 Library Calls and Polymorphism . . . 69
7.1.5 Other Synchronization Strategies . . . 69

8 Formalizing Shelters **71**

9 Implementing Shelters **75**
9.1 Shelter Registration and Waiting . . . 75
9.2 Optimizations . . . 77

9.2.1 Important Optimizations . . . 78
9.2.2 Other Optimizations . . . 78
9.2.3 Proposed Optimizations . . . 79
9.3 Limitations . . . 79

IV Evaluation **81**

10 Evaluation of SharC **83**
10.1 SharC Evaluation . . . 83
10.2 Conversion Effort . . . 85
10.3 SharC with Groups . . . 85
10.3.1 Benchmarks . . . 86
10.3.2 Scaling . . . 87
10.3.3 Sources of Overhead . . . 88
10.3.4 The Need for Groups . . . 89

11 Evaluation of Shelters **91**
11.1 Experimental Setup . . . 91
11.1.1 Intel STM . . . 92
11.2 Benchmarks . . . 92
11.2.1 Effects of Workload Size . . . 97
11.2.2 Discussion . . . 97

12 Conclusion and Future Work **99**
12.1 Future Work . . . 99

Bibliography **101**

V Appendix **107**

A The Soundness of SharC with Groups **109**
A.1 Preliminaries . . . 109
A.2 Runtime Typing . . . 109
A.3 Consistency . . . 110
A.3.1 Basic Properties . . . 112
A.3.2 Preserving Consistency . . . 114
A.4 Preserving Type Safety . . . 116
A.5 Thread Creation and Destruction . . . 117
A.6 Soundness Proof . . . 119

B The Soundness of Shelters **121**

C Concurrent Reference Counting **123**
C.1 Overview 123
C.2 Reference Update 124
C.3 Reference Count Calculation 125

List of Figures

2.1 The structure of the Ivy compiler. 14
2.2 Deputy annotation example 15
2.3 Deputy instrumentation example 16
2.4 Relationships between the components of Ivy. 20

4.1 A simple multi-threaded pipelining scheme 32
4.2 The annotated stage structure 34
4.3 Code that defines a linked-list structure 35
4.4 Code that uses SharC's single object cast 36
4.5 Annotated linked-list example for Groups 37
4.6 Our linked list example shown after instrumentation by SharC 40
4.7 An oct-tree example 42

5.1 A simple imperative language 46
5.2 A simple group 46
5.3 Typing judgments for definitions, types, and lvals. 49
5.4 Typing judgments for statements 50
5.5 Small steps in the operational semantics. 51
5.6 Runtime checks. 52
5.7 Operational semantics. 52
5.8 Auxiliary functions for the operational semantics. 53

6.1 The procedure for checking a sharing cast. 58

7.1 Shelters code example 65
7.2 Shelters example with instrumentation 66

8.1 Traces of shelter-based programs. 71
8.2 Trace Operational Semantics 72

9.1 Psuedo-code for the shelter register function. 76
9.2 Psuedo-code for the shelter wait function. 77

10.1 Scaling of SharC 88
10.2 The breakdown of overheads in SharC 89

11.1 Benchmarks for shelters 93
11.2 Benchmarks for shelters 94

A.1 Elided rules 110
A.2 Runtime typing judgments for expressions. 110
A.3 Runtime typing judgments. 111
A.4 Runtime checks for assignments. 112
A.5 Initial State 118

C.1 The procedure for updating a reference. 124
C.2 The procedure for calculating reference counts. 126

List of Tables

10.1 Benchmarks for SharC 84
10.2 Benchmarks for SharC with Groups 86

11.1 Shelter benchmark statistics 95

Acknowledgments

I thank my advisor Eric Brewer. His encouragement, technical insights into systems programming, and enthusiasm for the Ivy project were integral to the work in this dissertation. I also thank George Necula, whose admonition to “speak up!” early in my grad student career proved invaluable, and for his guidance in the realm of programming languages.

I am also very grateful for the mentoring provided by David Gay during internships and otherwise at Intel Labs, Berkeley. SharC would not exist if I had not benefited from his technical experience, cleverness, and meticulous attention to detail. I would also like to thank my other collaborators at Intel Labs, namely Rob Ennals and Mayur Naik. Countless discussions with David, Rob, and Mayur have doubtlessly improved the work in this dissertation. I am also grateful to everyone at Intel Labs, Berkeley for their hospitality, computing resources, well-stocked kitchen, wonderful lunch companions, and the most beautiful view a research lab could possibly hope for.

I am also indebted to various research groups in our CS department. Not long after my arrival at Berkeley, Adam Chlipala invited me to attend the lunch meetings of the Open Source Quality group, which introduced me to much of the amazing work in PL and compilers being done by grad students here. This led to my joining the Ivy group. In particular, collaborating with Jeremy Condit, Matt Herren, Feng Zhou, and Bill McCloskey on building and using Deputy was invaluable to the research in this dissertation. Finally, I have had the privileged and pleasure of subjecting the very patient members of the ParLab OS group to the Ivy compiler, and from this experience I have gained a working knowledge of how to avoid annoying systems programmers. For this experience, and their feedback, I thank Barret Rhoden, Keven Klues, David Zhu, Andrew Waterman, Paul Pearce, and NanWan.

I likely would not have enjoyed my time in grad school nearly as much without the the support and friendship of my dear `ucb_chums`. I have shared innumerable games of Settlers, world tours in Rock Band, lunches at Biryani House, pitchers at Triple Rock, hiking trips, enjoyable conversations, and weekend barbecues with: Ben and Juliet Rubinstein, Alex Simma, Louis Alarcon, Leon Barrett, Subbu and Praveena Venkatraman, Bonnie Kirkpatrick, Lauren Barth-Cohen, Anu Bowonder, and Pratik Patel.

Finally, I would like to thank Christina and my parents for their unwavering love and support.

Part I

Introduction

Chapter 1

Introduction

On the Feynman Problem-Solving Algorithm:
(1) write down the problem;
(2) think very hard;
(3) write down the answer.

MURRAY GELL-MANN

The ongoing migration of mainstream processors to many cores accelerates the need for programmers to write concurrent programs. In other words, in order for advances in hardware to continue giving improvements in efficiency, new software will have to be written such that performance improves as cores are added. Additionally, programs written with uniprocessor systems in mind may need to be refactored to take advantage of the heretofore unexpected direction in hardware evolution.

Threads are a popular mechanism for writing concurrent programs with the C programming language, using the pthreads API. Because threads consist only of a set of registers, a stack, and a program counter, they are cheap for operating systems to create, maintain, and destroy. Further, because multiple threads share the same virtual address space, sharing program data structures among threads is as inexpensive as sharing a memory address. In multi-threaded programs, access to such shared data structures is often mediated by locks, which ensure mutually exclusive access; and condition variables, which cause a thread to wait until a condition on the program state is met and a signal is received. These primitives synchronize threads so that each may acquire exclusive access to objects in turn without corrupting program state.

1.1 Unintended Sharing

Unfortunately, programmers generally find concurrent programming with threads much more difficult than sequential programming. One significant reason for this is that unintended data sharing among threads can make program behavior hard to understand or predict. By *unintended sharing* we mean data shared among threads in ways that the programmer does not expect and that are not part of the design of the program. Data sharing could be unintended if, for example, the programmer believes that a thread has exclusive access to an object when it does not. Such

situations arise when appropriate locks are not acquired, when they are not acquired in the right place in the program, or when condition variables are used with the wrong condition, or signals are sent or received at the wrong time.

One key symptom of unintended sharing is a *data race* — when two or more threads access the same memory location without synchronization, and at least one access is a write. Consider the following example in which two threads attempt to increment a counter, `count`, at the same time by reading the value of the counter into a local temporary before writing the result back to the counter:

Thread 1:	Thread 2:
`tmp1 = count;`	
	`tmp2 = count;`
	`count = tmp2 + 1;`
`count = tmp1 + 1;`	

In this example, there is a data race between Threads 1 and 2 because each reads and writes `count` without any synchronization. There exist interleavings of the statements that result in the counter being correctly incremented twice, however, nothing prevents an interleaving, like the one shown, in which the increment by Thread 2 is lost. This data-race occurred because the programmer assumed that `count` would not be shared during the increment. That is, the sharing during the increment was *unintended sharing*.

This kind of unintended sharing is often considered a major source of hard-to-find bugs in systems-level multi-threaded C programs. Searching for vulnerabilities caused by race conditions in the US-CERT advisory archive yields hundreds of results [89]. The unintended sharing that causes data races is difficult to diagnose because its occurrence and its effects are highly dependent on the way that the execution of threads is interleaved by the operating system's scheduler. This fact may frustrate developers using traditional tools such as a debugger or print statements, which may alter the schedule. Even when unintended sharing is benign, it may indicate places the programmer may need to investigate more carefully.

1.2 Deadlock

In addition to unintended sharing of data among threads, concurrent programming with threads and locks is difficult because the misuse of locks can result in a situation called *deadlock*, in which threads can no longer make progress. Lock-based deadlock arises when there is a cycle in the waits-for graph. For example, if thread T_1 holds lock L_1, and thread T_2 holds lock L_2, and if T_1 then waits for lock L_2, and T_2 waits for L_1, neither thread will be able to make progress.

Although deadlock is considered a less insidious bug than a data-race[1], the two are closely related. A programmer may add locks to avoid data-races, but doing so incorrectly may lead to deadlock. Consider the following function in which funds are transfered between two bank accounts.

[1] Since the program grinds to a halt, there is less concern about mysteriously corrupted program state.

```
void transfer(account_t *to, account_t *from, float amount) {
  to->balance += amount;
  from->balance -= amount;
}
```

Now, consider that accounts are shared among threads, and that one thread calls `transfer(A,B)`, and that another thread calls `transfer(B,A)`. Without any synchronization, this situation is clearly a data-race. As above, if the statements run by the threads are interleaved unfortunately, money may be generated or destroyed by lost writes. Fixing this problem is not as straightforward as adding locks, say `to->lock` and `from->lock`, and acquiring them at the start of the `transfer` function. This will result in a deadlock given our transfers between accounts `A` and `B` above because one thread acquires `A->lock` and then tries to acquire `B->lock`, while the other thread acquires `B->lock` and then tries to acquire `A->lock`, creating a waits-for cycle.

Some more sophisticated locking scheme is required to avoid this problem. Such sophisticated schemes are difficult to get right, and errors in them are likely to be in the form of further data-races and deadlocks. Given these difficulties, as one might expect, deadlocks are a big problem in large software projects. Indeed, as of March 2010, according to their bugzilla databases, there were 62 known, outstanding race conditions in the Linux kernel, and 23 in Firefox; and there were 30 known, outstanding deadlocks in the Linux kernel, and 46 in Firefox [56, 67].

1.3 The C Programming Language

Due to the high degree of control and performance that it affords, programmers use the C language [57] for writing low-level systems software. That it is difficult to write concurrent programs in C is not a problem in and of itself. Modern languages deal with the above mentioned difficulties through innovations in language design, programming interfaces and runtime environments. However, due to the amount and complexity of existing C code, rewriting all concurrent C code in modern languages is likely infeasible,[2] and the absence of a modern replacement for C having the same advantages ensures that C will continue to be used to create new software for the foreseeable future.

Since its introduction in 1972, C has been used to write operating systems (e.g. UNIX), database systems (e.g. MySQL), network services (e.g. Apache), and is involved in many aspects of the critical infrastructure of our society. [3] In addition to systems-level programs, C is also used for portable application development. As of December 2009, about 36% of the 158,332 projects hosted on SourceForge, a popular open source project hosting website, included code written in C or C++. Furthermore, according to the website `langpop.com`, programmers with knowledge of C are the second most sought-after in help-wanted ads on `craigslist.com`; the C language is the 5th most numerous topic for books on programming sold by Powell's Books; and C is the

[2] In his novel *A Deepness in the Sky*, science fiction author and Computer Scientist Vernor Vinge hypothesizes that, in the distant future, Programmer-Archaeologists will be widely employed.

[3] Additionally, C is used for the programming of embedded systems. In particular, the nesC [44] language, which is based on C, and which compiles to C code, is widely used in academia and industry for the programming of networks of sensor motes.

most used language for projects listed on Google Code, Freshmeat, and Oholo. C is also the most discussed language. It is mentioned most frequently on the Lambda the Ultimate blog, second most on `programming.reddit.com` and `slashdot.org`, and the fourth most on the Freenode IRC network. Clearly, projects written in C continue to be developed, and C continues to be used for many new projects due to the niche it fills.

Accordingly, some effort is clearly needed to address the shortcomings of C with respect to concurrent programming as multi-core processors become more prevalent. However, depending on the techniques used, it may first be necessary to address other shortcomings of C, in particular the lack of type- and memory-safety. The rest of this chapter is organized as follows. First we will give an overview of the efforts toward type- and memory-safety undertaken in the Ivy compiler. This will be followed by an overview of the extensions to Ivy, namely SharC and Shelters, designed and implemented to address the difficulties of concurrent programming in C, and which are the topic of this dissertation. Finally, we will give an overview of our evaluation of SharC and Shelters, which includes the application of SharC and Shelters to several large, widely-used applications written in C.

1.4 The Ivy Compiler

The goal of the Ivy compiler is to provide an evolutionary pathway from C to a language with stronger safety guarantees. In light of the difficulties mentioned above, jumping directly to a new language is likely infeasible. Therefore, the design philosophy of Ivy is to provide ways for the programmer to transition software in a modular fashion from C to a language with the desired safety guarantees. In practice, this means that the programmer must be able to make the transition to a safe language one compilation unit at a time, and also one safety property at a time. This design decision imposes two requirements on Ivy. First, code compiled with Ivy must interoperate seamlessly with code compiled by an off-the-shelf C compiler like gcc [38]. Second, the analyses used by Ivy to provide safety guarantees must not rely on having the whole source code of a program available. That is, if module M of a program is compiled with Ivy, but module N is not, the safety guarantees provided by Ivy to module M should not be invalidated. This does not preclude the possibility that module N may use module M incorrectly. It only requires that when module M is used correctly, Ivy's guarantees will hold.

Given these requirements, Ivy provides memory-safety to sequential programs with two components, one called Deputy [22], and the other called Heapsafe [43]. Deputy provides *spacial memory-safety*. That is, assuming that there are no dangling references, Deputy guarantees that a program makes no out-of-bounds memory accesses. Heapsafe provides *temporal memory-safety*. That is, it guarantees that there are no dangling references.

Deputy allows the programmer to make annotations on pointer types that describe the size of the region of memory referenced by the pointer. It makes use of the observation that information describing the bounds of the region have usually already been included in the program. Therefore, Deputy's annotations are *dependent type* attributes, which allow the types of pointers to depend on the runtime values of the expressions describing the bounds of memory regions. Deputy uses this information to construct runtime assertions that ensure that pointers are always in-bounds. The use of dependent types by Deputy, which make use of information already in the program and

require no changes to data layout, allow Ivy to retain both the interoperability and modular safety guarantees stipulated by its design philosophy. If pointers are always in bounds, and there are no dangling references, then the Deputized parts of a program are known to be memory-safe.

Heapsafe uses reference counting of pointers to check that no references to a memory region remain when memory is deallocated. That is, it checks that there are no dangling references. Similar to Deputy, Heapsafe requires no changes to data layout, and so code processed with it remains interoperable with other C code. Further, since the reference counting is largely automatic, requiring no interaction from the programmer, applying Heapsafe in a modular fashion is not a concern. It is necessary, however, to assume that libraries for which source code is not available are have no effect with respect to reference counts.

The Deputy and Heapsafe components of Ivy are sound for sequential code. However, both are unsound when faced with multi-threaded code. In particular, Deputy would suffer from time-of-check/time-of-use issues, while Heapsafe's reference counting would suffer from data-races if pointers are updated concurrently. In addition to making multi-threaded programming in C easier, the integration of SharC and Shelters into the Ivy compiler addresses the soundness issues that Deputy and Heapsafe have in multi-threaded programs.

1.5 SharC

As mentioned above, data-races are difficult to locate and to debug. However, we view data-races as a symptom of unintended sharing. Therefore, our general approach to eliminating data-races is to consider most sharing erroneous, even if it is mediated by locks or condition variables, *unless* it has been explicitly declared by the programmer.

This dissertation presents SharC [6, 7], pronounced "shark", a tool that allows a C programmer to declare the data sharing strategy that a program should follow, and then uses static and dynamic analysis to check that the program conforms to this strategy. Dynamic analysis allows SharC to check any program, including programs that would not be easily checkable with purely static analysis. SharC's static analysis improves performance by avoiding dynamic checks where unintended sharing is impossible.

When using SharC, a programmer uses type qualifiers to declare the *sharing mode* of objects. Sharing modes can declare that an object is thread-private, read-only, protected by a lock or shelter, intentionally accessed in a racy way, or checked dynamically. The dynamic sharing mode checks at run time that the object is either read-only, or only accessed by one thread. This allows SharC to check programs that would be difficult to check with a purely static system. The annotation burden is low because SharC makes reasonable and predictable guesses at unannotated types. Further, SharC allows the programmer to change an object's sharing mode using a cast, and uses reference counting to check the safety of such casts dynamically (Chapter 4).

We show that SharC's mixed static/dynamic checking approach is sound (i.e., detects all data races and checks all sharing mode changes), for a simple C-like language with thread-private, dynamically-checked, and lock-protected sharing modes (Chapter 5). We then describe SharC's implementation, in particular how it selects sharing modes for unannotated types and implements its runtime checks (Chapter 6).

In addition to the existing sharing modes and sharing cast of SharC, we have developed a new concept that we call *groups* by borrowing ideas from region, ownership, and dependent type systems. A *group* is a collection of objects all having the same sharing mode. Each group has a distinguished member called the *group leader*. When the sharing mode of the group leader changes by way of a sharing cast, the sharing mode of all members of the group also changes. We distinguish these casts from SharC's single-object casts by calling them *group casts*. We ensure soundness of group casts by requiring that all external pointers into a group have a type that *depends* on a pointer to the group leader, and using reference counting to prevent casts when more than one pointer to the group leader remains. Objects can be added or removed from groups, and group casts can be used to combine groups. We present the syntax along with examples of group use in Chapter 4, and formalize and prove the soundness of our group type system in Chapter 5. Our formalism for groups is not strongly tied to the concept of a sharing mode, and could be used to track other properties, such as tainting, immutability, etc. We hope that groups and group casts represent a promising approach to describing the runtime evolution of such properties.

SharC can be applied to programs incrementally. As more annotations are added, the false positive rate drops and performance improves.

Our implementation also includes an adaptation of the fast concurrent reference counting algorithm found in the work of Levanoni and Petrank [62]. We describe our modifications, as they may also be useful in memory management schemes for C that rely on reference counting and wish to handle multi-threaded programs efficiently.

1.6 Shelters

Though explicit locking can yield highly efficient code, its use is prone to errors such as data-races and deadlocks. Though SharC can find such data-races, it offers no assistance in rewriting code to eliminate them, or in remedying deadlocks. Atomic sections are a convenient language construct for controlling access to shared state in multi-threaded programs. They ensure that the statements within them execute atomically. That is, the effects of statements in atomic sections only become visible to other threads all at once when execution leaves the atomic section, much like a data base transaction. Because they simply declare that code is to be run atomically, rather than fully specifying how code is to be made atomic, the use of atomic sections is less error prone than explicit locking.

As multi-core processors have become more prevalent, newly developed languages have begun to include atomic sections instead of explicit locking [3, 18, 28]. Furthermore, attempts have been made to add atomic sections to pre-existing languages such as C and C++ [20, 51, 66, 80]. Atomic sections may be implemented either optimistically, as in software transactional memory(STM) [85] systems, or pessimistically, as in Autolocker [66].

STM systems execute atomic sections optimistically, at least in part. Atomic sections are allowed to execute concurrently, but when two or more threads make conflicting accesses, transactions must be rolled-back and retried. Many STM implementations achieve good performance. However, if transactions are large, or if data is highly contended, roll-backs may be frequent and expensive. Furthermore, roll-back may not be possible if, for example, any I/O was performed during a failed transaction.

The relative merits of optimistic and pessimistic concurrency control have been investigated by the database community. The consensus of this work seems to be that optimistic approaches are desirable in the presence of abundant resources, so that the cost of roll-back/replay is not significant, whereas pessimistic approaches are desirable when resources are scarce [2]. In the future, when resources may become abundant, production quality STM systems may overcome these practical difficulties. In the meantime, while resources are scarce, a pessimistic implementation of atomic sections can avoid these problems, while giving comparable performance in the cases where STM performance does scale.

In this dissertation we also focus on a pessimistic method for implementing atomic sections for C. Previous approaches to this problem have relied on whole-program analysis for determining a global lock order [66], or for calculating aliasing information used to construct fine-grained lock hierarchies [20, 51]. But whole-program analysis is often problematic in practice. First, it is often expensive for large programs. Second, the source code for the whole program may not always be available. We avoid these problems by using a mechanism we call *shelters* [5] to implement atomic sections. Our implementation is inspired by the deadlock-free synchronization strategy used in the Jade programming language [79].

Like explicit locks, shelters are first-class objects in our extension of C. Programmers use a `sheltered_by(s)` annotation on types to indicate that concurrent access to an object is mediated by the shelter `s`. Upon entry into an atomic section, threads must "register" for all of the shelters protecting the objects that are accessed in the atomic section. In order to register for a set of shelters, a thread atomically acquires a globally unique, increasing sequence number, and places itself on a queue for each shelter. Before accessing a sheltered object, a thread must then wait unless it is the thread on the shelter's queue having the smallest sequence number. When a thread exits an atomic section, it removes itself from the queues of the shelters for which it had registered. In Chapter 8 we show that this mechanism is sufficient for providing atomicity.

Using our system, programmers write code with `sheltered_by(s)` annotations and atomic sections. We discover for which shelters registration is required through a backwards dataflow analysis over atomic sections. We then use the results of the analysis to translate the program with atomic sections into a program using the register and wait calls mentioned above. Where the analysis is imprecise, we make use of coarser-grained shelters based on the types of the objects in question. An overview of our system's operation is given in Chapter 7 along with an example.

1.7 Evaluation

To determine the effectiveness of SharC and Shelters, we applied their features to a collection of application and benchmark programs. We ran SharC without groups on a set of C programs summing to over 600k lines of code. Two of the benchmarks were over 100k lines of code, and some contained kernels that had been finely tuned for performance. Further, we chose benchmarks that use threads in a variety of ways, from threads for serving clients, to hiding I/O and network latency, to improving performance. We found that for this set of programs, the number of annotations required to eliminate false positives and achieve reasonable performance (i.e., 9% runtime and 26% memory overheads) was small.

With the addition of groups, SharC was able to check a wider range of programs. In particular,

we were able to check data sharing in a set of programs including the GIMP image manipulation program, which is over 900k lines. Our benchmarks for SharC with groups include scientific codes using graphs and trees to organize calculations; a simple webserver using an in-memory cache shared among threads, which process requests from clients; and two image manipulation programs that use threads to improve performance and hide latency. We have observed that SharC with groups also has a manageable annotation burden. For small programs, around a few thousand lines, we made about one change for every 50 lines, but for the GIMP we only needed one for every 10,000 lines. Additionally, making the right annotations requires minimal program understanding — the GIMP required one person only three days to annotate. The scientific benchmarks had overheads around 40% (mostly due to concurrent reference counting), the overheads on the other benchmarks were less than 20%, and overheads were not affected by the number of threads.

We have also applied our shelter-based atomic section implementation to 11 programs including the STAMP benchmark suite [15], and a few representative programs totaling over 100k lines of code. On average, the runtime overhead incurred by shelter-based atomic sections remains between 0 and 20% with respect to explicit locking for programs using between 1 and 32 threads on a 32 core machine. We present a thorough comparison of the runtime and programming cost of shelter-based atomic sections with the Intel software transactional memory (STM) implementation [54] for the above-mentioned benchmarks.

1.8 Summary

In summary, the contributions of this dissertation are the following:

- A lightweight type annotation system that uses a mixture of statically and dynamically checked properties to describe many of the data sharing conventions used in existing multithreaded C programs. We show that in practice, these annotations are not overly burdensome. Indeed we believe that they are sufficiently simple and clear that they could be viewed as compiler-checked documentation.

- We show that dynamic reference counting makes it possible to safely cast an object between different sharing modes.

- We present *groups* and *group casts*, lightweight mechanisms for describing data structure properties, and evolution of these properties. Groups borrows ideas from region, ownership and dependent type systems to describe sets of objects with a common property, represented by a distinguished leader.

- We describe how groups can be used to specify the sharing strategy used in concurrent programs that use complex data structures.

- We present the design and implementation of Shelters, a pessimistic method for implementing atomic sections that requires no whole-program analysis or transactional memory.

- We describe our implementation of SharC and Shelters, and demonstrate their practicality by applying them to several large, real-world, C programs and standard benchmarks. On

these programs we observe low performance overheads and annotation burdens. We believe these overheads are low enough that our analysis could conceivably be used by working programmers building production systems.

- We describe the integration of SharC and Shelters into the Ivy compiler, complementing its ability to provide type- and memory-safety to sequential programs with the ability to supply these guarantees in addition to concurrency-safety to multi-threaded programs.

The rest of this dissertation is organized as follows. Chapter 2 describes in detail the Ivy compiler, and the relationships among its components. Chapter 3 discusses related work. Chapters 4 through 6 discuss the design, implementation, and soundness of SharC. Chapters 7 through 9 discuss the design, implementation, and soundness of Shelters. Chapters 10 and 11 discuss our evaluation of SharC and Shelters. Finally, Chapter 12 concludes and discusses future work.

Chapter 2

Ivy

> The whole is greater than the sum of its parts.
>
> *Metaphysica*
> ARISTOTLE

The previous chapter gave a brief overview of Ivy's design philosophy along with brief descriptions of its three main components. This chapter provides a more in-depth view of Ivy's components, and discusses how they work together to provide a collection of useful safety guarantees for multi-threaded systems-level programs.

Ivy can be applied in a modular, incremental way, and without the whole source code of a program available. This is accomplished through the use of type annotations on module boundaries, which can be supplied and checked one compilation unit at a time. Further we divide Ivy's functionality into independent components along the lines of related safety properties so that Ivy can be applied incrementally not only one module at a time, but also one safety property at a time.

Ivy compiles C with annotations to plain C, which can then be compiled by an off-the-shelf C compiler. Figure 2.1 shows Ivy's organization. A C source file with type annotations is passed to Ivy, which first uses Deputy to check for spacial memory-safety. This is followed by the Heapsafe and SharC components, which check for temporal memory-safety, and safe concurrency respectively. Components can be optionally disabled in which case they are simply bypassed. The output of Ivy is then built and linked with Ivy's runtime by the off-the-shelf C compiler. Because Ivy does not make changes to data-structures, and because it simply uses an off-the-shelf C compiler for its final compilation and linking steps, code compiled with Ivy remains interoperable with code compiled by an off-the-shelf compiler alone.

2.1 Deputy

Deputy provides type annotation for describing pointer bounds, tagged unions, null-terminated arrays, memory allocators and de-allocators, and some limited polymorphism. Deputy also strictly enforces C's type system. That is, it forbids casts between incompatible pointer types. The design of Deputy relies on the observation that C programs are already mostly correct, and that programmers already tend to include meta-data in a program sufficient for checking spatial memory-safety.

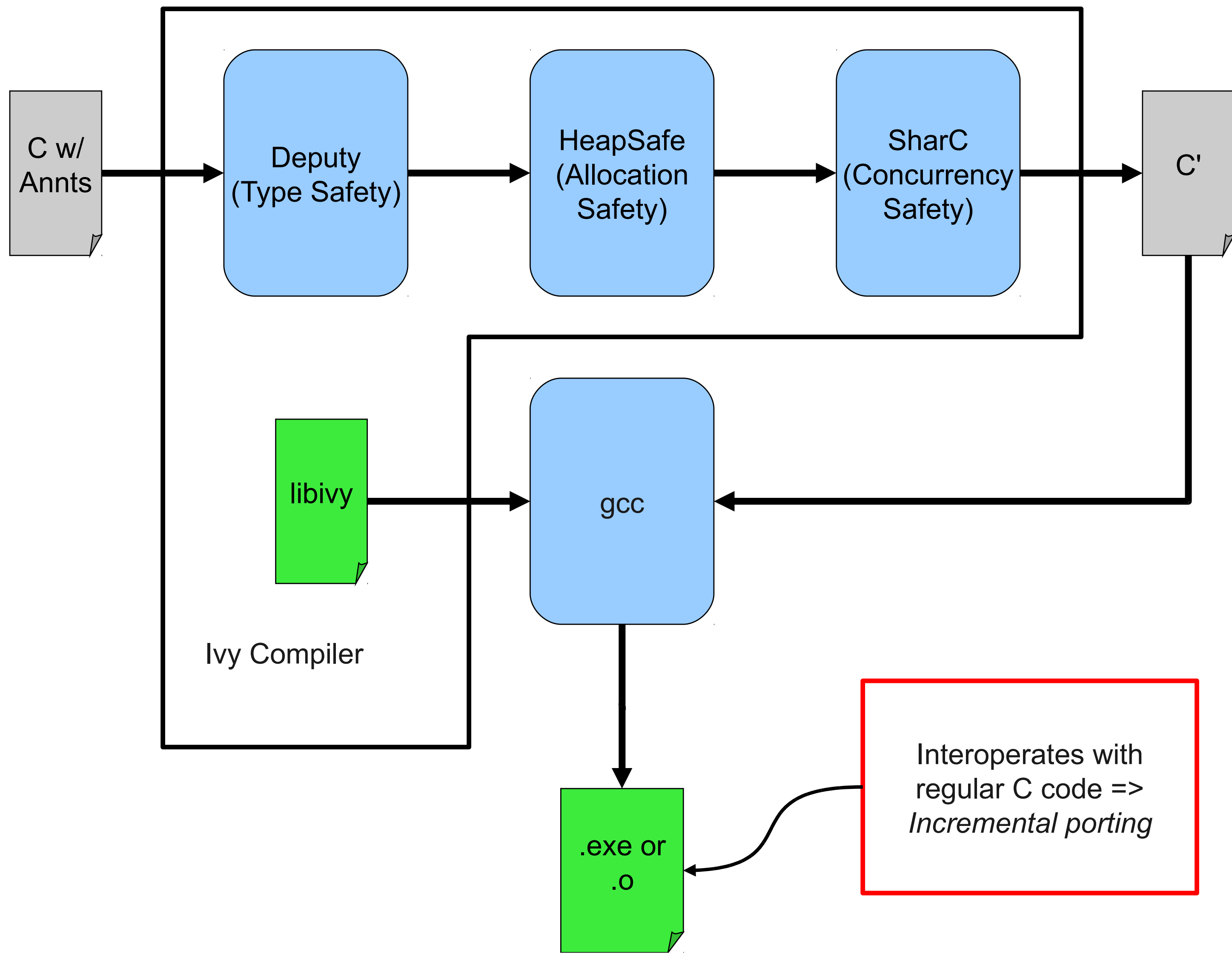


Figure 2.1: The structure of the Ivy compiler.

```
typedef struct {
  uint_t len;
  char * count(len) data;
} buf_t;

buf_t *init_buffer(char * count(len) data, uint_t len) {
  buf_t *newbuf = malloc(sizeof(buf_t));
  char *tmp = data;
  int i;

  for (i = 0; i < len; i++)
    *tmp++ = i;

  newbuf->len = len;
  newbuf->data = data;
  return newbuf;
}
```

Figure 2.2: An example of Deputy's annotations for a function that initializes a buffer.

Therefore, Deputy can check that pointers stay in bounds based on programmer annotations that refer to other variables, function parameters, and structure fields.

Deputy takes as input a partially annotated program, and operates in three phases. In the first phase, Deputy analyzes the existing annotations and the program. From this analysis it infers annotations on locals, and emits warnings when other declarations require explicit annotation. In the second phase, Deputy adds to the program assertions based on the annotations on types that are sufficient to ensure spatial memory-safety. In the third phase, Deputy attempts to identify assertions that will either always succeed or always fail. The former are removed, and the later are flagged as an error.

Consider the example in Figure 2.2. The structure type `buf_t` contains a buffer pointer, `data`, and an integer, `len`, indicating the size of the buffer. The pointer field has the Deputy annotation `count(len)`, which is shorthand for the more general annotation `bound(__self,__self+len)`. This indicates that `data` points to a buffer beginning at its current address value and extending for `len` elements. When `data` is non-null, and `len` is non-zero, Deputy forbids updates to `data`, and increments of `len`. (Updates of `*data` are still permitted.) Permitting either would prevent Deputy from disallowing dereferences of invalid addresses. However, it is still possible to decrement `len` down to zero. The annotation on the `data` parameter of `init_buffer` has the same meaning. These rules are specified in detail in Deputy's type system [22].

The local variable `tmp` requires no programmer annotation. Deputy infers the annotations on local variables that require them. Deputy's inference stage is borrowed from CCured [70]. That is, if arithmetic is performed on a pointer, then it is marked as needing an annotation if it does not already have one. These marks propagate through assignments. If the inference stage results in a local variable being marked, then Deputy creates auxiliary variables that store the bounds for the

```
1  typedef struct {
2    uint_t len;
3    char * bound(__self,__self + len) data;
4  } buf_t;
5
6  buf_t *init_buffer(char * bound(__self,__self+len) data, uint_t len) {
7    buf_t *count(1) newbuf = malloc(sizeof(buf_t));
8    assert(newbuf);
9    char *tmp_lo = data;
10   char *tmp_hi = data + len;
11   assert(tmp_lo <= data < data + len);
12   char *bound(tmp_lo,tmp_hi) tmp = data;
13   int i;
14
15   for (i = 0; i < len; i++) {
16     assert(tmp_lo <= tmp + 1 < tmp_hi);
17     *tmp++ = i;
18   }
19
20   assert(!newbuf->data || newbuf->data + len <= newbuf->data + newbuf->len);
21   newbuf->len = len;
22   assert(!data || data + newbuf->len <= data + len);
23   newbuf->data = data;
24   return newbuf;
25 }
```

Figure 2.3: An example of Deputy's annotations for a function that initializes a buffer.

pointer.

Consider the code fragment in Figure 2.3. This is the code with inferred annotations and assertions that Deputy emits after its second phase. Deputy infers that `tmp` needs an annotation. Deputy then creates local variables `tmp_lo` and `tmp_hi`, and annotates `tmp` with `bound(tmp_lo,tmp_hi)`. Then, before `tmp` is assigned, `tmp_lo` is assigned the lower bound of the assigned pointer, and `tmp_hi` is assigned the upper bound of the assigned pointer. In the example above, when `tmp` is assigned `data`, `tmp_lo` is assigned `data`, and `tmp_hi` is assigned `data+len`. When `tmp` is incremented, there is no need to reassign these bounds variables, as this is only necessary when the assigned value has a different bounds annotation. Further, since the type of `tmp` does not refer to itself, it is possible to increment it so long as it stays within `data` and `data+len`. No arithmetic is performed on the `newbuf` pointer, so it receives the default annotation of `count(1)`.

In the second phase, Deputy also adds assertions. In particular, it adds assertions that check the return of memory allocation functions (line 7), pointer assignments (lines 11 and 22), and pointer arithmetic (line 16). An assertion is also needed when a value on which a pointer type depends is assigned (line 20).

In its final stage, Deputy attempts to find statically assertions that will either always succeed,

which are removed, or always fail, which are bugs in either the program or the annotations. For example, in the assertion on line 16 above, Deputy can determine statically that the lower bound check is unnecessary because the pointer is only ever incremented.[1]

Deputy has been applied by itself to small benchmark suites including SPEC 95 [86], Olden [16], Ptrdist [8], and MediaBench [61]; to TinyOS [52], an operating system for embedded systems; and to a small (about 400k lines) Linux kernel. Overhead incurred by Deputy's runtime checks were under 100% for the small benchmarks and under 50% for Linux kernel benchmarks. On small benchmarks, applying Deputy required making changes on up to 30% of the lines, but on the Linux kernel changes were required on fewer than 0.6% of the lines.

Because Deputy uses meta-data already included in the program it does not need to change the layout of data structures. This feature allows it to both interoperate with other C code compiled by an off-the-shelf compiler, and to be applied to programs one file at a time, satisfying Ivy's design requirements.

In single-threaded programs without dangling references, Deputy enforces memory- and type-safety. The existence of dangling references breaks these guarantees. In addition to pointing to an invalid range of memory, consider that the memory referred to by a dangling reference may be reallocated to an object of a different type, thereby violating type-safety. A data-race in a multi-threaded program may also violate spacial memory-safety. That is, between the time that a Deputy assertion passes, and the time that the assertion is required to be true, another thread may do something that invalidates the assertion. For example, another thread may replace a pointer to a buffer with a pointer to a smaller buffer. Both of these issues are addressed by combining Deputy with Heapsafe and SharC

2.2 Heapsafe

Heapsafe performs reference counting to prevent dangling references. It does this by requiring that the argument to a memory deallocation call is the last pointer into the memory region being deallocated. Heapsafe transforms a program to keep reference counts. In particular, it replaces the standard `malloc` library calls with versions that perform the appropriate checks, and instruments pointer updates with updates to reference counts. Reference counts are stored in a single, statically allocated array. If an operating system allocates pages to processes lazily, then a program using Heapsafe only uses additional memory for this array for the portions of it that are used.

Heapsafe requires programmer interaction in the following cases. First, the programmer may need to null-out pointers to eliminate false reports of bad deallocations. Second, the programmer may need to provide special "adjust" functions that indicate how to update reference counts for the fields of a union. That is, Heapsafe does not make assumptions about which union fields are active, and this information must be supplied by the programmer. Finally, Heapsafe includes "delayed-free scopes" for handling the deallocation of structures containing pointer cycles. Delayed-free scopes allow delaying checking of reference counts and memory deallocation until after all of the objects in a cycle have been visited and their pointer fields nulled-out.

[1]Deputy's static analysis is not currently sophisticated enough to remove the upper-bounds check, which must also check for overflow in the pointer arithmetic.

Heapsafe has been applied by itself to the SPEC2000 [48] and SPEC2006 [49] benchmark suites, which includes a perl interpreter, totaling over 500k lines of code. Runtime overhead averaged 11% while the memory overhead averaged 13%. On these benchmarks applying Heapsafe required making changes to 0.56% of lines on average.

The vast majority of Heapsafe's transformations require no programmer interaction, so it is inexpensive to apply it to a whole program all-at-once. Even though this lack of modularity seems to violate Ivy's design philosophy, since the lack of modularity imposes only a small cost on the programmer, we overlook it in favor of Heapsafe's many advantages. Furthermore, Heapsafe can soundly use external libraries so long as they retain no references to reference counted pointers. In general, this requirement violates Ivy's interoperability requirement, however libraries that operate like this for which the source code was not available did not occur in any of the millions of lines of code used to benchmark Ivy's components. Since program designs on which Heapsafe violates Ivy's design philosophy are rare, we again overlook this drawback of Heapsafe.

While ensuring the absence of dangling references, Heapsafe relies on the absence of nonsensical casts and out-of-bounds pointer arithmetic. Furthermore, as written Heapsafe is not thread-safe. That is, reference counts may go wrong in the face of concurrent pointer updates. These issues are addressed by Deputy and SharC.

2.3 SharC and Shelters

SharC and Shelters are described in great detail in the coming chapters. For now, it suffices to say that if SharC signals no static or dynamic errors, then there are no data-races on a run of a program. Additionally, SharC extends the Heapsafe framework with support for concurrent reference counting. However, because it relies on type annotations, SharC can only make these guarantees if the programs it checks are type- and memory-safe. As explained above, these guarantees are provided by Heapsafe and Deputy when they have the guarantees that SharC makes.

Shelters are included in Ivy as a sharing mode of SharC. That is, the programmer may specify that an object is protected from concurrent access by a particular shelter, and this status may change as a program executes by way of SharC's sharing cast. In a C program, if locks are replaced by Shelters and atomic sections, then Ivy can guarantee the absence of lock-based deadlock.

2.4 Putting It All Together

Figure 2.4 shows the guarantees provided and used by the different components of Ivy. SharC provides race-freedom, and relies on the type- and memory-safety provided by the combination of Heapsafe and Deputy. Deputy provides strict type-checking and spatial memory-safety, and relies on the race freedom and temporal memory-safety provided by SharC and Heapsafe, respectively. Heapsafe provides temporal memory-safety, and requires Deputy's type-checking and guarantees of spatial memory-safety to provide type-safety. Heapsafe also relies on the implementation of concurrent reference counting that SharC provides in the Heapsafe framework.

Together these tools provide the following safety guarantees:

- Spatial memory-safety (Deputy)

- Temporal memory-safety (Heapsafe)
- The absence of nonsensical type casts (Deputy)
- Type-safety (Deputy, Heapsafe)
- Data-race freedom (SharC)
- The absence of lock-based deadlock (Shelters)

On the other hand, no guarantees are made about the following properties:

- The absence of memory leaks
- The absence of non-lock-based deadlock

A typical Java runtime environment also provides some of these guarantees. Spatial memory-safety exists due to the prohibition on pointer arithmetic, and array bounds are checked at runtime. Temporal memory-safety is enforced by allocating objects only on the heap, and these objects are managed only by the garbage collector. The absence of nonsensical casts is enforced by dynamically checking an object's type. On the other hand, most Java runtime environments do no checking for data-races and deadlocks, though there are research prototypes that do these things. Further, it is well known that memory leaks can still exist when heap objects are managed by a conservative garbage collector.

The solutions used by Java to enforce these safety properties are not immediately available in the implementation of a systems programming language due to the need for pointer arithmetic, manual memory management, and the stack-allocation of aggregates. Therefore, Ivy provides new technology so that the guarantees found in modern languages can be made for a systems programming language.

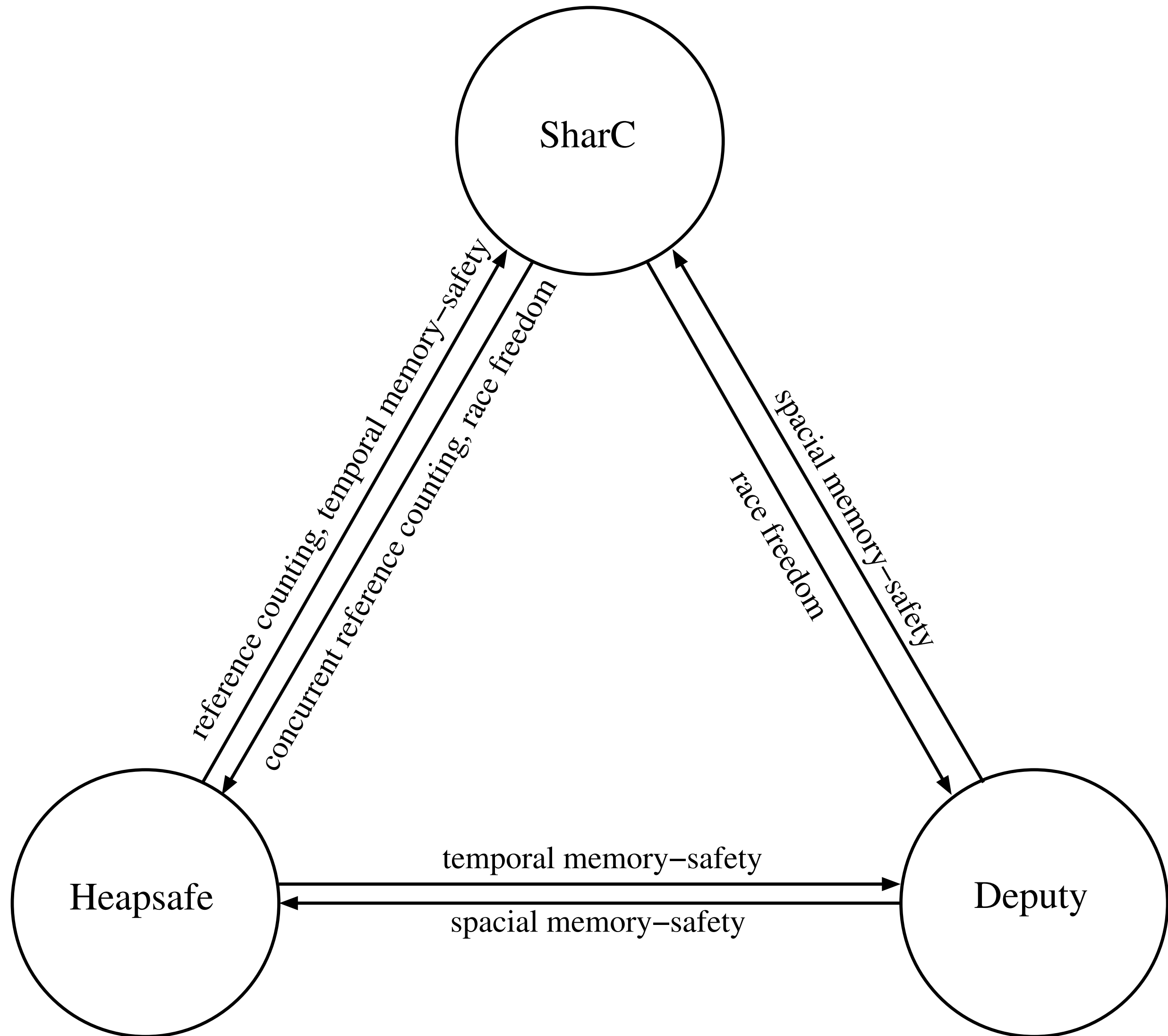


Figure 2.4: Relationships between the components of Ivy.

Chapter 3

Related Work

> There is, however, another purpose to which academies contribute. When they consist of a limited number of persons, eminent for their knowledge, it becomes an object of ambition to be admitted on their list.
>
> CHARLES BABBAGE

A number of attempts have been made to create safe, C-like languages for systems programming. These seek to enforce in systems programs the following safety properties: spatial memory-safety, temporal memory-safety, the absence of type-cast errors, and the absence of memory leaks. CCured [70] leaves the C language mostly unchanged.[1] It simply changes the representation of pointers so that spatial memory safety can be enforced. CCured also includes some other runtime type information where necessary for checking pointer casts, and uses the Boehm-Weiser garbage collector to enforce temporal memory-safety and to preclude some memory leaks. Cyclone [47] and Vault [25] make more extensive changes to the C language, and include features such as region-based memory allocation, and linear and unique pointer types.

As mentioned in a previous chapter, additional safety issues arise in concurrent C programs, namely data-races and deadlock. CCured does not provide protection from these problems. Cyclone and Vault have some mechanisms in their type-systems for preventing data-races. In particular, a linear pointer in Vault can be owned by only one thread at a time. It is a goal of Ivy to provide guarantees like those found in more modern languages, but in a way that requires less pervasive code changes and rewriting. To that end, this dissertation describes SharC and Shelters, which provide protection from data-races and deadlock, and which rely on and complete the type- and memory-safety guarantees provided by Deputy and Heapsafe.

Recently, work inspired by SharC, but relying completely on dynamic analysis in combination with API calls for changing sharing modes and ownership has been presented [65]. Other than this, however, the work most closely related to the work presented here are tools that attempt to find data races in concurrent programs, and tools that provide atomic sections in an attempt to avoid lock-

[1]CCured includes some annotations like Deputy's, which are not necessary for correctness, but whose use is encouraged for obtaining better performance and for documentation.

based deadlock. Whereas SharC attempts to identify violations of a sharing strategy, race detectors simply look for unsynchronized access to memory. There are several different approaches to race detection. We classify the approaches as static, dynamic, and model-checking. Additionally, many other researchers have investigated both type systems for describing aggregates of objects, and methods for using them in making concurrent programs safe. Finally, we describe various implementations of atomic sections for C and other programming languages.

3.1 Static Race Detection and Model Checking

There has been much work on statically analyzing data races and atomicity in Java programs [12, 69, 81]. We have used some ownership type ideas from these works in our own type system, and believe that atomicity is an important concern requiring further attention in dynamic analysis of large real-world legacy C programs.

The Chord system [68, 69] does not use type parameters, but relies on a precise alias analysis to show that distinct locks protect distinct sets of objects. This approach has found many bugs in real programs, but results in false positives when programs use signaling for synchronization, and when objects may be private to different threads in different phases of the program. The alias analysis does not distinguish program phases in which the same object is variously private or protected by a lock, for example. Further, this approach would not be suitable for legacy C programs due to the need for a precise whole-program alias analysis.

There has also been much work on statically analyzing atomicity in Java programs [35, 36, 81]. These use type- and effect- based analyses to ensure that the locks used to protect objects are actually sufficient for algorithmic correctness. Atomicity has race-freedom as a prerequisite.

Cyclone [47] allows the programmer to write type annotations that enable its compiler to statically guarantee race-freedom. The difficulty in applying Cyclone's analysis to existing multi-threaded programs lies in translation to a program with Cyclone's annotations and concurrency primitives. Our system requires annotations, but they are far less pervasive, they can be omitted from the program for compilation with an off-the-shelf C compiler, and the resulting binaries can be linked seamlessly with libraries compiled by an off-the-shelf C compiler.

Relay [91], and RacerX [30] are static lockset based analyses for C code that scale to large real world programs including the Linux kernel. However, both tools are unsound, and require significant post-processing of warnings to achieve useful results. Locksmith [74], on the other hand, is a sound static race detector for C. It statically infers the correlation of memory locations to the locks that protect them. If a shared location is not consistently protected by the same lock, a race is reported. Locksmith also does a sharing analysis similar to our own as an optimization. Unfortunately, Locksmith runs out of resources while analyzing larger programs.

Some of these techniques have scaled up to many hundreds of thousands of lines of code and have uncovered serious problems in real world software. Further, some of these techniques, especially the ones for Java, also achieve manageable false positive rates. For development cultures in which programmers are encouraged to use the results of static analysis, these techniques are probably an appropriate choice. However, testing is already widely used, and so low overhead dynamic analysis will be the least-cost path to race detection for many people. Additionally, there

is currently no static tool with low annotation burden, which has also been used on real programs, that captures the idiom of objects being private to different threads in different program phases.

Sen and Agha [83] perform explicit path model checking of multi-threaded programs by forcing races detected on a concrete run to permute by altering the thread schedule, and by solving symbolic constraints to generate inputs that force the program down new paths. KISS and the work of Henzinger et al. can also find errors in concurrent programs with model checking [50, 75].

We also wish to mention a few other related projects. First, race freedom can be checked by translation to a linear program based on fractional capabilities [87]. Since linear programming instances can be solved in parallel, this technique may be able to scale to large programs. However, the possibly substantial collection of warnings may be difficult to investigate because analysis results do not indicate what locks may be held at each program point. SharC scales to large programs, and gives detailed error reports.

Some other tools are also guided by programmer annotations. LockLint [26] is a static system that checks that locks are used consistently to protect program variables. It uses annotations to suppress false warnings, to describe hierarchical locking, and procedure side-effects, and as documentation. Further, the Fluid Project [46] has investigated a system for Java that allows a programmer to make annotations describing hierarchical regions, aliasing intent, and a locking strategy that can apply to regions, among other annotations. SharC differs from these systems primarily because it allows the sharing mode for data structures to change through sharing and group casts as the program executes.

3.2 Dynamic Race Detection

Initial work in dynamic race detection relied on Lamport's happens-before relation [60]. This approach is able to handle many different kinds of concurrency primitives including barriers, as it was developed by the scientific parallel programming community. However, it is difficult to implement efficiently, and the usefulness of the results are dependent on how threads are scheduled.

Eraser [82] popularized the dynamic lockset algorithm for race detection. The goal of the lockset algorithm is to ensure that every shared location is protected by a lock. Eraser monitors every memory read and write in the program through binary instrumentation, and tracks the state of each location in memory. The states that a location can inhabit model common idioms such as initialization before sharing, read-sharing, read-write locking, and so forth. Eraser is able to analyze large real-world programs, but it incurs a 10x-30x runtime overhead. Further, the state diagram used to determine when a race might be occurring may not be an accurate model of the data sharing protocol in a program. This inaccuracy leads to false positives.

Improvements to the lockset algorithm add states to capture more idioms, and use Lamport's happens-before relation to track thread-ownership changes [19]. This reduces false positives due to the Eraser lockset state diagram failing to model signaling between threads, among other things. Further lockset algorithms have been developed for handling barrier synchronization [73]. Some dynamic race detectors perform preliminary static analysis to improve performance. Analyses using these improvements have achieved lower false positive rates and better performance [1, 33]. For Java, the overhead has been reduced to 13%-42% [21]. Goldilocks integrates race detection into the Java runtime [29]. Racetrack integrates race detection into the CLR [93], and

achieves overhead in the range 1.07x-3x for .Net languages, with the low end corresponding to non-memory-intensive programs. Preliminary dynamic analyses for deciding which objects to monitor for races have also been attempted [72], but this approach requires reproducing the inputs to a program. Using the happens-before relation and more complicated state diagrams to model additional data sharing schemes reduces false positives, but our system is the first to attack the root of the problem by modeling ownership transfer directly.

3.3 Ownership and Region Type Systems

Ownership types have been used to statically enforce properties such as object encapsulation, race-freedom, and memory safety [11]. An ownership type system statically guarantees that every object has a unique owner and that the ownership relation is a forest of trees. The type system allows multiple pointers to an object but statically checks that only the pointer from its owner has special privileges. For instance, an ownership type system for object encapsulation prevents unrestricted accesses to an object by checking that all accesses happen via the pointer from its owner [13]. Likewise, an ownership type system for race-freedom requires each access to an object o to be protected by a lock held on its *root* owner, which is the object at the root of the tree in the ownership relation forest that contains o [12]. Finally, an ownership type system can be combined with a region type system and used for region-based memory management [14] in which each object is allocated in the same region as its owner. The resulting type system is more restrictive and enforces memory safety by forbidding pointers from outside the region to objects inside the region which enables it to safely free all objects in the region whenever the region handle is freed. Further, dynamic analyses have been developed to help programmers visualize ownership relationships in their programs [77]. In these systems, instead of requiring programmers to annotate instances of encapsulation, which is the common case, the programmer is shown the cases in which encapsulation is violated, which is typically less common.

The relationship of a group leader to objects in its group is analogous to that of an owner to objects it owns. Groups, however, are more general in that objects can be added to or removed from them and group leaders can change dynamically, in contrast to the ownership relation, which cannot be changed dynamically. We have found that this ability is crucial for applying an ownership-like system to large C programs.

Region type systems have strong similarities to groups: they use types to identify a collection of objects, and allow global operations (typically deallocation) to be performed on that group. RC [42] is the closest to our groups: it uses reference counting to track pointers into regions, and has a `sameregion` qualifier to identify pointers that stay within a region. However, as with other regions systems, region membership is fixed at allocation time and tracked explicitly at runtime, regions cannot be merged and `sameregion` is mostly a performance hint to avoid the need for reference-counting. Tofte and Talpin's statically checked region type system [88] places further restrictions on region lifetimes and pointers between regions: essentially, region lifetimes are nested and pointers are only allowed from longer-lived regions to shorter-lived regions, but not vice-versa. Crary and Walker's [23] use of capabilities relaxes these restrictions somewhat, but still does not allow unrestricted pointers between regions. Effectively, SharC's use of reference-counting allows a runtime-check to recover the linearity property of a group (region) reference

that the static type systems must ensure at compile-time.

The authors of Cyclone [47] note the similarity in their type system between types for region membership and types for lock protection. Indeed, they go so far as providing a shorthand for when objects in the same region are protected by the same lock. SharC's Groups expand on this idea by allowing objects belonging to the same data structure (which may be unrelated with respect to memory allocation concerns) to be related not only by locking, but also a variety of mechanisms for safe concurrency. Further, we note that SharC would likely benefit from some of the polymorphism in Cyclone's type system, but we leave this extension for future work.

3.4 Atomic Sections

Many researchers have investigated the implementation of atomic sections, and more generally, language constructs enabling correct concurrency. Fortress [3], Chapel [18], and X10 [28] are new languages intended to be used for writing highly scalable, high-performance code. Each of them includes atomic sections for protecting shared state.

A few projects are more closely related to our own. Autolocker allows programmers to annotate variables and fields as being protected by a particular lock [66]. Then, for each atomic section, it determines which locks must be acquired, and in what order they must be acquired. Determining lock order requires a whole-program analysis, which is often impractical for large projects written in C. Our system avoids the need to find a global lock order, and therefore allows C projects to retain separate compilation.

Cherem et al. [20] present a system in which the locks required for an atomic section are inferred from the structure of expressions accessed in the atomic section. The granularity of the locks may vary depending on the precision of the underlying analyses. Like Autolocker, this approach requires a whole program analysis for calculating pointer aliasing and for refining the expressions accessed in an atomic section. The approach of Hicks et al. [51] is somewhat similar, requiring a whole program alias analysis to acquire a coarse-grained lock for the possible targets of a pointer when a precise target cannot be determined. The coarsening strategy for our system is similar, however instead of performing a whole-program alias analysis, we simply assume that pointers of the same type may alias. A more precise analysis could be integrated into our implementation to obtain a finer-grained shelter hierarchy, however, for the above mentioned practical reasons, we chose to avoid whole-program analysis.

In the Jade [79] programming language, programmers make annotations to describe how concurrent tasks will access shared state so that the Jade compiler can then automatically extract concurrency. Access to the shared state is then mediated with a mechanism that inspired our implementation of shelter-based atomic sections. In particular, shared objects each maintain a queue of tasks waiting to access them. When many tasks attempt to access the same object, the task with the smallest sequence number is permitted to proceed. We use the shelter mechanism to enforce atomic sections in an explicitly parallel program rather than as a way to help a compiler extract parallelism in an implicitly parallel language. Our system's implementation also expands on this mechanism by introducing explicit shelter objects that allow the programmer to declare what objects need protection, eliminating the need for the programmer to make annotations at each atomic block, and by introducing a shelter hierarchy. Further, our lock-free implementation reduces the

extent to which programs are serialized by accesses to queues.

There also exist several transactional memory systems that can be used to implement of atomic sections, both hardware [4, 76] and software [39, 64, 84] based. We believe that the software transactional memory (STM) system for C most similar to our own in terms of programmer convenience is the Intel STM implementation [54]. Other STM implementations for C give better performance [15], but they require by-hand instrumentation of reads and writes of shared memory. The Intel STM implementation incurs overhead from the instrumentation of all shared memory reads and writes inside of transactions, and from the rollback of transactions during which conflicts are detected. Because our system is pessimistic, it does not incur these overheads. Furthermore, rollback may not even be possible if side-effects occur in third-party code, and is often an unreasonable approach for systems code. Our pessimistic approach avoids these problems. Boehm argues that transactional memory should be viewed as a mechanism for providing atomicity rather than a programming interface [10].

Type systems with annotations describing locking rules have been used to prevent data races [31, 32], in particular for Java [34, 11]. Boyapati's ownership type-system [11] allows the expression and checking of sophisticated locking schemes. Our system could be extended with similar ownership or region types as an alternate way to arrive at a finer-grained shelter hierarchy. We leave these extensions for future work. Also for Java, Hindman and Grossman [53] translate programs with atomic sections to ones that acquire locks just before they are needed. Deadlock is avoided at runtime through rollback.

Part II

SharC

Chapter 4

Design of SharC

> If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.
>
> *The Humble Programmer*
> Edsger W. Dijkstra

In this chapter, we give a detailed description of the design of SharC. First, we discuss the API it presents to programmers, and give examples of its use. Later in the chapter, we pay special attention to the feature of SharC, called Groups, which allows the programmer to describe sharing modes that apply to, and change across, complicated data structures.

4.1 Design of SharC

SharC uses a mixture of static and dynamic checking to provide an API that is simple to understand, and yet is complete enough that large real programs can be checked with reasonable runtime efficiency. SharC's first key concept is a set of *sharing mode* annotations that describe the rules that threads must follow when accessing a particular object. The programmer can annotate types with the following sharing mode qualifiers:

- `private`: Indicates that an object is owned by one thread and only accessible by that thread (enforced statically).
- `readonly`: Indicates that an object is readable by any thread but not writeable, with one exception: a `readonly` field in a `private` structure is writeable (enforced statically). This exception makes initialization of `readonly` structure fields practical, while maintaining the read-only property when the structure is accessible from several threads. Unlike C's existing `const` qualifier, `readonly` may be safely cast to another sharing mode that allows writes, as described below.

- `locked(`*lock*`)`: Indicates that an object is protected by a lock and only accessible by whatever thread currently holds the lock (enforced by a runtime check). Here, *lock* is an expression or structure field for the address of a lock. Since the `locked()` qualifier depends on a runtime value, it must follow similar dependent type-checking rules as those imposed by Deputy. In particular, if the value of an expression on which a `locked()` qualifier changes, then the pointer of the `locked()` type must be null. If the `locked()` type is not a pointer type, then its lock must be verifiably constant.

- `sheltered_by(`*shelter*`)`: Indicates that concurrent access to an object is mediated by `shelter`. Like `locked`, this is a dependent qualifier.

- `barrier(`*b*`)`: Indicates that SharC is to enforce at runtime that the object is either read-only, or only accessed by a single thread in between barrier calls on barrier `b`.

- `dynamic`: Like the `barrier` mode, but an object in this mode is required to be either read-only or accessed by a single thread for the duration of the program, not simply between barriers.

- `racy`: A mode for objects on which there are benign races (no enforcement required).

SharC takes as input a program with sharing mode annotations made on some of the types. Using a simple set of rules and an optional whole-program thread-escape analysis, SharC selects sharing mode annotations for the unannotated types. These rules simply propagate user-provided annotations in a straightforward way, and the thread-escape analysis allows SharC to infer the more efficient, `private` sharing mode for non-escaping objects. The rules and thread-escape analysis are described in detail in Chapter 6. In the examples below, we assume that the thread-escape analysis is used, and refer to SharC inferring (or not) that unannotated objects are thread-shared, and therefore in the `dynamic` sharing mode. Unannotated objects that do not thread-escape SharC infers to be in the `private` sharing mode. If the whole-program thread-escape analysis is not used, SharC assumes that any unannotated object may thread-escape, and places them in the `dynamic` sharing mode. The sharing modes inferred by the rules and the thread-escape analysis are not trusted; they are statically checked for well-formedness, and the inferred sharing modes are enforced by SharC's static and dynamic analysis.

The `dynamic` sharing mode makes SharC practical for large, real-world programs by avoiding the need for complex types, specifications, or precise whole-program analysis. However, in real programs, objects often go through a sequence of sharing modes. For example, in a producer-consumer relationship, an object is first private to the producer, then protected by a lock, then private to the consumer. Thus, SharC's second key feature is a *sharing cast* that allows a program to change a single object's sharing mode:

```
int readonly *y; ... x = SCAST(int private *, y);
```

SharC enforces the soundness of these casts by nulling-out the pointer being cast, and by using reference-counting to ensure that the pointer being cast is the *only* reference to the underlying object. If we have the only reference to some object, then we can, e.g., safely cast it to or from `private`, since no thread will subsequently be able to see the object with the old sharing mode.

SharC can infer where sharing casts are needed to make a program type-check. However, since nulling-out a cast's source may break the program, and since there may be several options of varying suitability for the location of a cast, SharC does not insert these casts automatically. Instead SharC suggests where they might need to be added. It is then up to the programmer to add the suggested casts to the program if they are safe, or make alternative changes if they are not. Additionally, SharC will emit a warning if a pointer is definitely live after being nulled-out for a cast.

4.1.1 An Example

Consider a multi-threaded program in which a buffer is read in from a file, passed among threads in a pipeline for various stages of processing, and then output to disk, screen, or other medium. This is a common architecture for multimedia software, for example the GStreamer framework [40].

The core function implementing such a pipeline is shown in Figure 4.1. The `stage` structures are arranged in a list, and there is a thread for each stage. Each thread waits on a condition variable for `sdata` (line 10). When data arrives, the pointer is copied into a local variable (line 20), the original pointer is nulled (line 21), and the previous stage is signaled (line 22). This indicates that the current stage has taken ownership of the data, and that the previous stage is free to use the `sdata` pointer of the current stage for the next chunk of data. Having claimed the data, the current stage processes it somehow (line 25), waits for the next stage to be ready to receive it (line 16), copies the pointer into the next stage (line 31), and then signals that the next stage may claim it (line 32). The process then repeats until all chunks have been processed.

SharC will compile this code as is. However, since the programmer has not added annotations to tell SharC the desired sharing strategy, SharC will assume that all sharing it sees is an error, and will generate an error report when sharing occurs. This is accomplished by SharC inferring the `dynamic` sharing mode for unannotated objects, and the dynamic analysis that enforces it. SharC reports two kinds of data sharing. First, it reports sharing of the `sdata` field of the `stage` structures. The following is an example of such a report.

```
read conflict(0x75324464):
  who(2) S->sdata  pipeline_test.c: 17
  last(1) nextS->sdata  pipeline_test.c: 31
```

This indicates that thread 2 tried to read from address `0x75324464` through the l-value `S->sdata` on line 17 of the file after thread 1 wrote to the address through l-value `nextS->sdata` on line 31. Without knowledge of the desired sharing strategy, SharC assumes that this sharing is an error.

This error report was generated by SharC's dynamic checker. SharC was not able to prove statically that the `sdata` field was private, so it inferred the `dynamic` sharing mode for `sdata`, and monitored it at runtime for races (two threads have accessed the same location, with at least one access being a write).

As a human reading the code, it is clear that the programmer intended the `sdata` field to be shared among threads, and to be protected by the `mut` lock. We can declare this policy to SharC

```
1  // pipeline_test.c
2  typedef struct stage {
3    struct stage *next;
4    cond_t *cv;
5    mutex_t *mut;
6    char *locked(mut) sdata;
7    void (*fun)(char private *fdata);
8  } stage_t;
9
10 void *thrFunc(void *d) {
11   stage_t *S = d, *nextS = S->next;
12   char *ldata;
13
14   while (notDone) {
15     // wait for the data pointer to be passed from the previous stage
16     mutexLock(S->mut);
17     while (S->sdata == NULL)
18       condWait(S->cv,S->mut);
19     // copy the data pointer into a local
20     ldata = SCAST(char private *, S->sdata);
21     S->sdata = NULL;
22     condSignal(S->cv);
23     mutexUnlock(S->mut);
24     // now that this stage has the pointer, we can run the stage's function
25     S->fun(ldata);
26     // if there is a subsequent stage, then pass the data pointer on
27     if (nextS) {
28       mutexLock(nextS->mut);
29       while (nextS->sdata)
30         condWait(nextS->cv,nextS->mut);
31       nextS->sdata = SCAST(char locked(nextS->mut) *, ldata);
32       condSignal(nextS->cv);
33       mutexUnlock(nextS->mut);
34     }
35   }
36   return NULL;
37 }
```

Figure 4.1: A simple multi-threaded pipelining scheme as might be used in multimedia software. Items in bold are the additions for SharC.

by adding a `locked` annotation to line 6. Now, rather than checking that `sdata` is not accessed by more than one thread, SharC will instead check that the referenced lock is held whenever `sdata` is accessed. That is, an explicitly annotated type will lead to the enforcement appropriate for the sharing mode; SharC no longer places the object in the `dynamic` mode, or assumes that all sharing of the now-annotated object is an error.

SharC will also report sharing of memory *pointed to* by the `sdata` field. Here is an example:

```
write conflict(0x75324544):
  who(2) *(fdata + i)  pipeline_test.c: 52
  last(3) *(fdata + i)  pipeline_test.c: 62
```

Lines 52 and 62 are not shown in the figure, but are both in functions that can be pointed to by the `fun` field of the `stage` structure. In these functions, `fdata` is equal to the `ldata` pointer that was passed as the argument to `fun`. The error message indicates that thread 2 tried to write to address `0x75324544` through the l-value `*(fdata+i)` on line 52 after thread 3 had read from the same address through the l-value `*(fdata+i)` on line 62.

Here, SharC is not aware that ownership of the buffer is being transferred between the threads. It believes that the buffer is being shared illegally because the type of the buffer was unannotated, and so placed in the `dynamic` mode, and then at runtime, the buffer was accessed by more than one thread. The user can tell SharC what is going on by adding an explicit `private` annotation to the `fdata` argument of `fun` on line 7. This will cause SharC to infer that `ldata` is `private` rather than `dynamic`, but type checking will fail at lines 20 and 31 due to the assignment of a (`char locked(mut)*`) to and from (`char private*`). To fix this, SharC suggests the addition of the sharing casts (`SCAST`(...)) shown in bold on lines 20 and 31. As discussed above, these sharing casts will null-out the cast value (`S->sdata` or `ldata`) and check that the reference count for the object is one. For line 20 this ensures the object referenced by `S->sdata` is not accessible by inconsistently qualified types (`locked(...)` and `private`). These two annotations and two casts are sufficient to describe this simple program's sharing strategy, and allow it to run without SharC reporting any errors.

Figure 4.2 shows the sharing modes selected by SharC for the `stage` struct, and the first few lines of the `thrFunc` function. The `next`, `cv` and `fun` fields inherit the structure's qualifier `q`. The internals of pthread's lock and condition variable types (`mutex`, `cond`) have data races by their very nature, so they have the `racy` qualifier. The `mut` field must be `readonly` for type-safety reasons, as the type of `sdata` depends on it. The object referenced by `sdata` has "inherited" its pointer's sharing mode. The object referenced by `next` has not been annotated, so must be given the `dynamic` mode since the structure's qualifier can't be similarly "inherited" for referent types for soundness reasons, as discussed in the next section. By default, without the thread-escape analysis enabled, SharC infers the remaining types to be in the `dynamic` mode (except for non-function-escaping locals). With the thread-escape analysis enabled, SharC infers that the object passed to `thrFunc` is accessible from several threads, so `d`, `S` and `nextS` must be pointers to `dynamic`, and the remaining missing annotations become `private`.

At runtime for the annotated program, SharC will check that:

1. When the `sdata` field of a `stage` structure is accessed, the `mut` mutex of that structure is held.

```
1  // pipeline_test.c
2  typedef struct stage(q) {
3    struct stage dynamic *q next;
4    cond racy *q cv;
5    mutex racy *readonly mut;
6    char locked(mut) *locked(mut) sdata;
7    void (*q fun)(char private *private fdata);
8  } stage_t;
9
10 void dynamic*private thrFunc(void dynamic *private d){
11   stage_t dynamic *private S = d;
12   stage_t dynamic *private nextS = S->next;
13   char private *private ldata;
14   ...
15 }
```

Figure 4.2: The `stage` structure, with the annotations inferred by SharC shown un-bolded. The qualifier polymorphism in structures is shown through the use of the qualifier variable `q`.

2. When the non-`private` pointer `S->sdata` is cast to `private` on line 20, no other references to that memory exist.

3. When the `private` pointer `ldata` is cast to a non-`private` type on line 31, no other references to that memory exist.

4. There are no races on objects inferred to be in `dynamic` mode.

4.2 Limitations of the Single-Object Sharing Cast

With the capabilities of SharC mentioned so far, programmers are able to describe the sharing strategy in a variety of legacy multi-threaded C programs. Unfortunately, the restriction that the sharing cast may apply only to single objects prevents SharC from being easily applied to a number of common code patterns.

In this section, we will refer to a simple example in which a singly-linked list is initialized by one thread before being shared through a lock with other threads. The code for this example is given in Figure 4.3. This code snippet defines a type for list nodes, and gives a function that initializes a list before sharing it with other threads through a global variable protected by a lock.

The sharing strategy for this code dictates that the list is private when being constructed, and then becomes lock-protected when it is shared with other threads on line 20. We would like to simply declare that `lckL` is a pointer to a `locked() list_t`, and that its `next` field also points to `locked() list_t`'s, and that similarly `pL` is a pointer to a `private list_t` with a `next` field pointing to `private list_t`'s. However, if we do that, as so far described, SharC will not let us do a sharing cast from `pL` to `lckL` at line 20. In the program as written, without some other mechanism, it would

```
1  typedef struct list {
2    int data;
3    struct list *next;
4  } list_t;
5
6  mutex_t *Lck;
7  list_t *lckL;
8
9  void init() {
10   list_t *pL = NULL, *end = NULL;
11   for (int i = 0; i < 10; i++) {
12     list_t *t;
13     t = malloc(sizeof(list_t));
14     t->data = i;
15     t->next = NULL;
16     if (pL == NULL) { pL = t; end = pL;}
17     else { end->next = t; end = t; }
18   }
19   mutex_lock(Lck);
20   lckL = pL;
21   mutex_unlock(Lck);
22 }
```

Figure 4.3: Code that defines a linked-list structure, and shows how it is initialized by one thread before being shared through a lock with other threads.

```
lcklist_t locked(Lck) *convert(plist_t private *L) {
   plist_t *pt1 = L, *pt2;
   lcklist_t locked(Lck) *lckhp = NULL,
   lcklist_t locked(Lck) **lcktp = &lckhp;

   while (pt1) {
     pt2 = pt1->next;
     pt1->next = NULL;
     *lcktp = SCAST(lcklist_t locked(Lck) *, pt1);
     lcktp = &(*lcktp)->next;
     pt1 = pt2;
   }
   return lckhp;
}
```

Figure 4.4: Code that uses SharC's single object cast to convert a `private` list to a `locked()` list. `SCAST` is SharC's single object sharing cast.

be unsound for SharC to allow such a cast because there is nothing to prevent a private pointer into the tail of the list subsequently pointed to by `lckL`.

Alternately, we might try to constrain ourselves to the features of SharC mentioned so far, and define two different list structures, each with the appropriate annotation given explicitly on the `next` field, say `plist_t` and `lcklist_t`. Then, we could traverse the list, doing a single-object sharing cast on each node while casting from one list node type to the other.[1] We might write something like the code in Figure 4.4. This approach has a few problems. First, we would have to compel SharC to accept the dubious cast between two different aggregate types, which it currently does not. Second, code with a similar goal may become quite cumbersome to write for anything beyond the simplest data structures. Finally, for data structures consisting of many individual objects, the above code would adversely affect run-time performance. Furthermore, condoning such an approach would violate SharC's design goal of requiring the programmer to write only simple type annotations and casts to specify a program's sharing strategy.

4.3 Groups

To address this issue, SharC includes a feature called *Groups*. The goal of Groups is to allow changing the sharing mode, not only of individual objects, but also of complex data structures in one atomic action. In order to ensure the soundness of these operations, we require the programmer to add annotations that describe a *group* of objects that have the same sharing mode. A group is identified by a distinguished object called the *group leader*. Internal and external pointers to the group have distinct types such that external pointers must identify the group leader. We combine static and runtime checks to maintain the invariant that no object is referred to by pointers with

[1]This tactic might require writing to `readonly` objects, if that is the intended target mode, and so will trivially violate SharC's type system, but suppose for a moment that this is not a problem.

```
1  typedef struct list {
2    int data;
3    struct list same *next;
4  } list_t;
5
6  mutex_t *Lck
7  list_t group(lckL) locked(Lck) * locked(Lck) lckL;
8
9  void init() {
10   list_t group(pL) private * private pL = NULL;
11   list_t group(pL) private * private end = NULL;
12
13   for (int i = 0; i < 10; i++) {
14     list_t group(pL) private * private t = NULL;
15     t = malloc(sizeof(list_t));
16     t->data = i;
17     t->next = NULL;
18     if (pL == NULL) {pL = t; end = pL}
19     else {end->next = t; end = t;}
20   }
21   mutex_lock(Lck);
22   lckL = GCAST(list_t group(lckL) locked(Lck) *, pL);
23   mutex_unlock(Lck);
24 }
```

Figure 4.5: Code that defines a linked structure, and shows how it is initialized by one thread before being shared through a lock with other threads. Bolded annotations are provided by the programmer. Unbolded annotations are inferred.

different sharing modes at the same time. In brief, we ensure that all external pointers identify the same object as the group leader, and that group casts are only performed when there is a single external pointer to the group. The rest of this section explains the syntax for groups, and how our type system enforces the above semantics.

4.3.1 Group Annotations and Casts

Groups are specified using two annotations. First, we provide a dependent type annotation, `group(p)`, which indicates that objects of that type belong to the group with leader pointed to by `p`. Second, we provide an annotation for structure field pointer types, `same`, which indicates that the structure field points into the same group as the containing structure. The `group(p)` annotation identifies *external* pointers, while the `same` annotations identifies *internal* pointers. We also provide an additional checked cast operation, `gcast`, which indicates that the sharing mode or group leader for a group of objects is changing. `gcast` ensures that there is only one external reference to a group leader, and atomically copies the value of the pointer into the new pointer with the new type, and nulls out the old pointer so that no pointers with the old type refer to the

same data structure after the cast.

Consider the code snippet in Figure 4.5. This is our linked list example from above, but now it is fully annotated with sharing modes, and casts. Bolded annotations must be provided by the programmer, while the unbolded annotations are inferred. First note the `same` annotation in the type of the `next` field on line 3. This indicates that `next` is an internal pointer in the same group as the enclosing structure. Next, note that `lckL` is a `locked()` pointer into a `locked()` list. The `group(lckL)` annotation indicates that `lckL` is an external pointer into the group identified by the target of `lckL`. That is, `lckL` is a pointer to the group leader. Inside of the `init()` function, `pL` and `t` are `private` pointers to `private` lists. They are both external pointers into the group identified by the target of `pL`. In the `for` loop, nodes are added to the list.

After the list is built, on line 22, the mode for the entire list is changed from `private` to `locked()`. This is safe because `pL` is the only external reference to the list, and because no other live, non-null variable refers to `pL` in its type. These checks are sufficient to ensure that there are no pointers with the wrong type. If our list building code had, e.g., stored an external reference into the middle of the list, then its type would have to refer to the group leader. If the type referred to a reference aside from `pL`, then the reference count on the group leader would be too high. If its type referred to `pL`, then the null check on pointers that refer to `pL` would fail.

4.3.2 The Group Leader

As mentioned, when a cast of a group is attempted, we must be able to verify that there is only one external pointer into the group. Further, for soundness reasons when a pointer mentioned in a dependent type is assigned to, pointers of types that depend on it must be null.

To enforce the first requirement, we restrict what expressions may appear as parameters to a `group()` annotation. In the annotation `group(p)`, `p` is a pointer expression built from variables and field names: the types of structure fields may refer to other fields of the same structure, and the types of locals may refer to other locals (including formal parameters) in the same function. Globals may be mentioned in the `group()` parameter, but only when they are declared `static readonly`. Finally, pointers mentioned in the `group()` parameter may not have their addresses taken. These restrictions permit us to reason locally about what values types depend on. That is, we require no alias analysis to discover what the dependencies of a type are. These are the same restrictions as those used in the Deputy [22] dependent type system for C.

The second requirement exists to ensure that the group of an object cannot change without a group cast. That is, changing the group of an object by simply changing the value of the pointer in its `group()` annotation is forbidden. To enforce this, we simply insert dynamic checks for null-ness ahead of assignments to group leader pointers on which other live pointers depend. Note that in our linked list example in Figure 4.5, on line 18 `t` is dead after the assignment, so no null-check is required for `t`. However, `pL` is live after the assignment, and so we must check that it is null beforehand. In short, pointers whose type depends on `pL` must be null before an assignment to `pL` or dead afterwards. These restrictions may seem cumbersome, however they have not caused any substantial difficulties in porting our benchmarks.

In a fully typed program, every pointer type has either a `group()` annotation or a `same` annotation. However, the `group()` parameter may be null. When this is the case, the object cannot

be the subject of a group cast as it clearly cannot be the group leader, however it can be extracted from the "null" group using the single-object sharing cast as described in Chapter 5. In our system we use the shorthand nogroup for group(NULL). In our linked list example, the nogroup annotations would go on types that do not have a group() or same annotation, but these have been omitted for readability. In our implementation nogroup is the default annotation. This is so that the programmer is required to make annotations only for instances of data structures where the group cast is needed.

4.3.3 Operations on Groups

Objects can be added to and removed from groups, while group casts can be used to change a group's sharing mode and/or merge two groups. Consider the linked list structure above. Adding a node to the list is straightforward. We simply declare a new list node pointer, and indicate that it points into the needed group as shown in the linked list example.

In our first example with list_t, we demonstrated the use of the group cast to change the sharing mode of an entire linked list. This idiom is fairly common in concurrent C programs. That is, a data structure is initialized privately to one thread before being shared among all threads either protected by a lock, or read-only. We can also merge one group into another group. This operation requires a checked cast because we do not currently support any notion of "subgroups," and so no pointers identifying the leader of the old group as their group leader may exist after the merge:

```
list_t group(Ltail) private * private Ltail;
list_t group(Lhead) private * private Lhead;
// ...  nodes are added to Ltail ...
Lhead->next = GCAST(list_t group(Lhead) private *, Ltail);
```

Due to the few restrictions placed on internal group pointers (i.e. the same pointers), our type system does not support group splitting. However, we do support the removal of individual objects from a group using the single-object sharing cast; if the same fields of an object are null, then it can be cast to any group. The inability to split groups has not been problematic in our benchmarks.

4.3.4 Instrumentation

In Figure 4.6 we show the instrumentation that SharC uses for its dynamic analysis. In fact, this snippet is simply meant to show logically what the instrumentation does since the actual implementation involves some amount of optimization to reduce the cost of reference counting, and some additional complexity because the reference counting must be thread safe. Since this example involves only the private and locked() modes, all of the instrumentation comes from checking that Lck is held when lckL is accessed (line 29), reference counting, and checking the reference count at the group cast (line 31). Since we can verify statically that t is dead at the assignment at line 18 of Figure 4.5, and that t and end are dead at the group cast at line 22 of Figure 4.5, the only dynamic checking needed for the group() types is ensuring that pL is null before it is assigned (line 10). If t and end were not determined statically to be dead at the above mentioned locations, the instrumented code would contain assertions requiring them to be null.

```
1  void init() {
2    list_t *pL = NULL, *t = NULL, *end = NULL;
3    for (int i = 0; i < 10; i++) {
4      decrc(t);
5      t = malloc(sizeof(list_t));
6      incrc(t);
7      t->data = i;
8      t->next = NULL;
9      if (pL == NULL) {
10       assert (pL == NULL);
11       decrc(pL);
12       pL = t;
13       incrc(pL);
14       decrc(end);
15       end = pL;
16       incrc(end);
17     }
18     else {
19       decrc(end->next);
20       end->next = t;
21       incrc(end->next);
22       decrc(end);
23       end = t;
24       incrc(end);
25     }
26   }
27   mutex_lock(Lck);
28   sharc_lock_acquire(Lck);
29   assert(sharc_has_lock(Lck));
30   // The following six lines are performed atomically
31   assert(refcount(pL) == 1);
32   decrc(lckL);
33   lckL = pL;
34   incrc(lckL);
35   decrc(pL);
36   pL = NULL;
37   sharc_lock_release(Lck);
38   mutex_unlock(Lck);
39 }
```

Figure 4.6: Our linked list example shown after instrumentation by SharC. Since the example is using only the `private` and `locked()` modes, the only instrumentation is for reference counting, checking that locks are held, and checking the group cast.

4.3.5 A More Complex Example

Figure 4.7 gives pseudo-code for an n-body simulation using the Barnes-Hut algorithm [9]. At each simulation time-step, Barnes-Hut constructs an oct-tree based on the locations of the bodies, then computes the center of mass of each node in the oct-tree. The force on each particle is computed by walking the oct-tree; the effect of all particles within an oct-tree node can be approximated by the node's center of mass as long as this is far enough from the particle. Finally, the computed force is used to update a particle's velocity and position.

Figure 4.7 omits the code that builds the tree and does the calculations to highlight the data structures, sharing modes and group casts. This pseudo-code has been adapted to pthreads from a Split-C [58] program (itself inspired by the Barnes-Hut Splash2 [92] benchmark) written in an SPMD style; it uses barriers to synchronize accesses to shared data structures, and each thread has a unique id given by `MYPROC`. The tree is built out of `node_t` structures. Each `node_t` records the total mass of the bodies below that node and the position of their center of mass. Further, leaf nodes in the tree can point to up to eight bodies, while internal nodes have eight children. The bodies are represented in the `node_t`'s as an index into a `body_t` array whose elements represent the particles. Finally, every node has a pointer to its parent node.

The function `run()`, executed by each thread, shows an outline of how the simulation runs. First, the initial positions and velocities of the bodies are written into the array of `body_t` structures(line 24) by one of the threads. This initialization occurs privately to this thread. This is reflected by the `private` sharing mode of the `bodies` array on line 21.

Next, the threads enter the main loop. First, an oct-tree is built by the same thread that initialized the bodies(line 28). While the tree is being built, the parent pointers in the nodes are filled in. In the last step of `BuildTree()`, the tree is walked up from each leaf node to calculate the mass and center of mass for each node[2]. During this process the parent nodes are nulled out, as they will not be used again.

After the tree is built, it will no longer need to be written. Further it will need to be read by the other threads. This sharing mode change is implicit in the original program through the use of the `barrier()` call, but with SharC such changes must be explicit. Hence, we use a group cast to make the entire tree `readonly` (line 29). This cast succeeds because there is only one external reference to the whole oct-tree, and because no other pointers aside from `root` mention `root` in their types. Further, no other pointers aside from `roRoot` mention `roRoot` in their types, so it is also safe to write `roRoot`. If `BuildTree` had saved an external reference into the tree, then this external reference would have been dependent on some pointer to the group leader. This unsafe condition would then be caught during the checked cast because the reference count to the group leader would be greater than one.

We also cast the `bodies` array (line 30). We use the `barrier` sharing mode because different threads "own" different parts of the array. The thread that owns a body calculates its new position, velocity, and acceleration based on the masses of nearby bodies (distant bodies are not accessed as their effects are summarized in the oct-tree). Some parts of the body structure are `readonly`, and other parts are `private`. This fact cannot be represented by our static type system, but we can

[2] We note here that it is sometimes necessary when using groups to pass an unused pointer to the group leader to recursive functions. This has not caused problems in our benchmarks.

```
1  typedef struct node {
2    double mass;
3    double pos[3];
4    struct node same *parent;
5    int bodies[8];
6    struct node same *children[8];
7  } node_t;
8
9  typedef struct body {
10   double mass;
11   double pos[3];
12   double vel[3];
13   double acc[3];
14 } body_t;
15
16 void BuildTree(node_t group(root) private *root, body_t private *Bodies);
17
18 void run() {
19   node_t group(root) private *root;
20   node_t group(roRoot) readonly *roRoot;
21   body_t private *bodies;
22   body_t barrier *barBodies;
23
24   if (MYPROC == 0) initBodies(bodies);
25   while (not_done) {
26     if (MYPROC == 0) {
27       root = malloc(sizeof(node_t));
28       BuildTree(root,bodies);
29       roRoot = GCAST(node_t group(roRoot) readonly *, root);
30       barBodies = SCAST(body_t barrier *, bodies);
31     }
32     barrier();
33     // Share roRoot and barBodies, then calculate new body positions.
34     barrier();
35     if (MYPROC == 0) {
36       root = GCAST(node_t group(root) private *, roRoot);
37       bodies = SCAST(body_t private *, barBodies);
38       free_tree(root);
39     }
40   }
41 }
```

Figure 4.7: An oct-tree is built privately to one thread, and then the entire tree is cast to `readonly` to be shared with all other threads.

check the desired access pattern using the `barrier` sharing mode. The cast of bodies can use the single-object sharing cast (`SCAST`) as the `body_t` type contains no `same` pointers.

Following these casts, the pointers to the body array and the tree root are shared through globals with the other threads, and the simulation is updated. Subsequently, these pointers are nulled out in all threads but one, and more casts are used to indicate that the body array and oct-tree are private again (lines 36 and 37). The oct-tree can then be destroyed (line 38).

If this program compiles and runs without errors with SharC, then we know that no pointers exist to the oct-tree or to the bodies with the wrong type, that there are no data races on the bodies while they are protected by the barrier synchronization (or on other objects in other sharing modes), and that only one thread accesses the oct-tree or the bodies while they are in the `private` mode.

On this example, SharC's dynamic checking incurs a runtime overhead of 38% when running a simulation of 100 time steps for approximately 16000 bodies. Most of this overhead is due to the cost of reference counting.

4.3.6 Limitations of Groups Design

We have found that Groups are sufficient for handling a number of common multi-threaded C programming idioms. In particular, they are effective when shared objects needing mode changes belong to only one group at a time. Said differently, our Groups type-system allows an object to belong to more than one data structure, but only when each of the data structures belongs to the same group. This is because we do not currently support simultaneous membership of objects in multiple groups. Such a feature might be useful if, for example, an object is stored in different data structures, and has different protection mechanisms depending on which data structure it is accessed through.

This restriction has not presented problems in converting our benchmarks, however restructuring a program to comply with the above restriction might reduce the amount of concurrency in a program, affecting performance. That is, it may be necessary to restructure a program so that access to an object is mediate by the more restrictive protection mechanism, even though this might not be required for correctness. Furthermore, even this solution may be inadequate if it is not possible to identify a group leader for all of the data structures to which an object may simultaneously belong.

Additionally, while it is possible to remove individual objects from groups in our system, it is not possible to split one group into two separate groups. This is because our current system cannot verify that formerly-internal pointers from one group to another do not exist after the split.

This restriction may be problematic, for example, in a program in which threads traverse a shared tree structure, and after traversing deeply enough, wish to treat a sub-tree as thread private. In this case, the programmer would like to do a group cast of the sub-tree to the private mode, but our system currently forbids such a cast. We believe we could address this shortcoming by keeping separate reference counts for internal and external pointers, or by using the results of a shape analysis.

Despite these limitations, we believe that Groups are a good way to describe the sharing of data structures among threads in many C programs.

Chapter 5

Formalizing SharC

> The best of ideas is hurt by uncritical acceptance,
> and thrives on critical examination.
>
> *How to Solve It*
> George Pólya

The purpose of formalizing our system is to gain confidence that it does indeed provide the guarantees advertised in the preceding sections. In particular, by formalizing our extensions to a language that models the essential features of C, we find that our approach is capable of ensuring that the sharing strategy encoded by the programmer with our annotations is indeed enforced by our implementation.

5.1 The Soundness of SharC with Groups

The formal model presented here is a combination of the formal models used for SharC [6] alone and SharC with Groups [7]. Our language (Figure 5.1) consists of global variables ($x : \tau$), C-like structures ($t = \ldots$) and function ($f()\{\ldots\}$) definitions (we assume all identifiers are distinct). Our global variables are unusual: for each global variable $x : \tau$ a structure instance o_x of type τ is allocated at program startup, and newly created threads have a *local* variable $x : \tau$ that is initialized to o_x. This allows sharing between our threads while simplifying our semantics by allowing all variables to be treated as local.

Figure 5.2 shows typical type declarations for two variables `x` and `y`: a type τ of the form $m<x>\ t$ represents a pointer to the structure named t, that can be null. The sharing mode m is the sharing mode of the pointer's target; our formalization supports `private` (accessible to only one thread), `locked` (each structure protected by an implicit lock, Java-style), and `dynamic` (checked at runtime for data-races). As we described earlier, each group is identified by a distinguished *leader* object. In our type system, all pointer types must include the name x of a variable or field that points to the leader (the syntax allows for arbitrary lvalues, but these can only appear during type-checking and in the operational semantics). For instance, in Figure 5.2 `y` points to the leader for `x`'s target. In the rest of this section, we simply call `y` tx's leader. Note that the leader of `y` is

Program	P	$::=$	$\Delta \mid P; P$
Definition	Δ	$::=$	$x : \tau \mid t = (f_1 : \phi_1, \ldots, f_n : \phi_n)$
		$\mid$	$f()\{x_1 : \tau_1, \ldots, x_n : \tau_n; s\}$
Dependent Type	τ	$::=$	$m<\ell>\ t$
Field Type	ϕ	$::=$	$t \mid \tau$
Sharing Mode	m	$::=$	`locked` | `private` | `dynamic`
Statement	s	$::=$	$s_1; s_2 \mid$ spawn $f() \mid$ lock $x \mid$ unlock x
		$\mid$	$\ell := e$ [when $\omega_1, \ldots, \omega_n$]
		$\mid$	**wait** | **done**
L-expression	ℓ	$::=$	$x \mid x.f$
Expression	e	$::=$	$\ell \mid$ new $\mid$ scast $x \mid$ gcast x
Predicate	ω	$::=$	$oneref(x) \mid chkread(x) \mid chkwrite(x)$
		$\mid$	$m(x) \mid \ell_1 = \ell_2 \mid \ell =$ null
Identifiers	f, x, t		

Figure 5.1: A simple imperative language. Elements in bold are only used in the operational semantics.

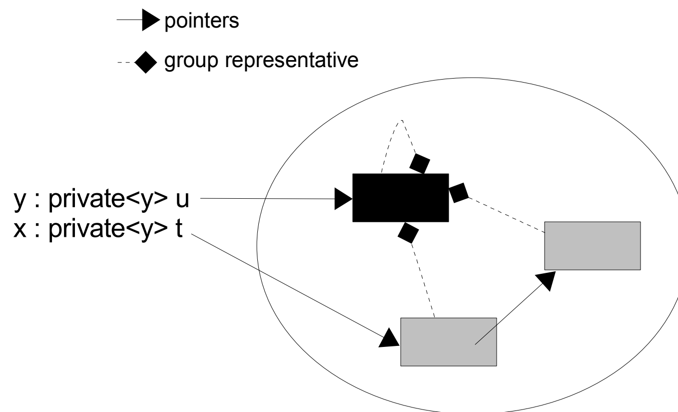


Figure 5.2: A simple group

y itself (this is required for soundness), however it is legal to have multiple pointers to the group leader. For instance, we could add declarations

```
y1 :  private<y1> u
x1 :  private<y1> t
```

and legally assign y to y1 and x to x1. Thus, the declaration (y : m<x> t) is equivalent to the C language declaration (t m group(x) * private y) since all variables in the formalism are locals. Our pointer types depend on variables and are thus a form of dependent types. To ensure soundness in the presence of mutation, we use similar type rules to the Deputy system [22] (see below).

Structures consist of a set of typed fields $f : \phi$, where ϕ is either a dependent type τ or an unqualified pointer to a structure t. In the first case, the type τ can only depend on other fields (as in Deputy) and cannot be private[1], in the second case the pointer is a "same group" pointer, i.e. its target is in the same group and has the same sharing mode as the structure itself. For example

```
list = (data:  locked<data> stuff, next:  list)
x :  private<x> list
```

declares a linked-list type where the list node structures are in a single group. Thus, these declarations are equivalent to the C language declarations:

```
struct list {
  stuff locked(data) group(data) *data;
  struct list same *next;
}
struct list private group(x) * x;
```

The variable x is thus a private list with locked contents: x, x.next, x.next.next, ... are all private objects in the same group, while the list contents x.data, x.next.data, ... are in separate groups protected by locks.

Functions f consist of local variable definitions and a sequence of thread-spawning, locking and assignment statements. Assignments are guarded by runtime checks (ω) that check sharing modes are respected ($m(x)$), compare lvalues ($\ell = \ell'$ and $\ell =$ null) and assert that x is the sole reference to a particular object ($oneref(x)$). These runtime checks are added automatically during type checking. The most important expressions are gcast x and scast x which perform a *group cast* on x's target, and a single object cast on x's target, respectively. For instance:

```
y :  locked<y> list
y := gcast x
```

casts our list x whose node structures were private into a list y whose node structures are protected by locks. A group cast can also merge two groups:

```
y :  locked<z> list
y = gcast x
z.next = y
```

[1] This requirement can be relaxed to forbidding non-private pointers to such structures.

A single list node, that is, a node with a null `next` field, can be cast from any sharing mode and group to any other sharing mode and group through the single object cast operation, scast. For example, suppose that the private list with head pointer (and group leader), `x`, has four elements:

```
xt :  private<x> list
y :  locked<z> list
xt = x.next.next.next
x.next.next.next = null
y = scast xt
```

extracts the tail of the private list, and casts it to a `locked` member of the group with leader `z`.

Our formalism can readily be extended with primitive types, and further sharing modes, for example Shelters would be treated much as locks are. However, our formalism does not currently support splitting a group into two arbitrary parts. Doing so would require precise tracking of intra-group pointers, which would substantially increase the complexity of our system.

5.1.1 Static Semantics

Figure 5.4 presents the type-checking rules for our language. These rules both check a program for validity and rewrite it to include necessary runtime checks. Also, we restrict assignments to the forms $x := e$ and $x.f := y$. To simplify exposition, we assume in the rules that $\Gamma_G(x)$ gives the type of global variable x, T is the set of all structure types, and $t(f)$ gives the type of field f of structure t.

The GLOBAL, STRUCTDEF and THREAD rules enforce the restriction that variable types are only dependent on other variables, while field types are only dependent on other fields of the same structure. Furthermore, globals cannot be `private` and structures cannot contain explicitly `private` fields, as having a non-private pointer to a structure with a private field would be unsound. The DEPENDENT rule requires that the leader of a group uses itself as leader.

Reading or writing lvalue ℓ (READ, WRITE) requires three sets of runtime checks. First, if $\ell = x.f$ we must enforce x's sharing mode. If x is in the `dynamic` mode, it is necessary to distinguish between reads and writes, which is accomplished through the R and W functions. Second, we must ensure that the lvalues denoting the assignment's source and target group match after the assignment. Third, the function D adds runtime checks to ensure that any variables or fields dependent on the assignment's target ℓ' are null before the assignment, as the assignment would otherwise be unsound. The scoping rules enforced in GLOBAL, STRUCTDEF and THREAD make these checks tractable: if $\ell' = x.f$, fields dependent on f must be in the same structure instance, while variables dependent on $\ell' = x$ must be variables of the same function. Finally, these checks remain sound even in the presence of concurrent updates by other threads: variables cannot be modified by other threads, while the runtime check ω that enforces $\ell = x.f$'s sharing mode also guarantees that any other fields of x can be accessed without races.

A group cast of y can be performed if y is the only reference to the target object. Because the group cast nulls-out y, the dependent-type restrictions (D) must hold for y, as in a regular assignment to y. A single-object sharing mode cast may also be performed if, in addition to the conditions for the group cast, the "same group" fiels are null.

$\vdash P \Rightarrow P'$	P' is identical to P except for runtime checks that ensure P's sound execution.

(DEFINITIONS)
$$\frac{\vdash \Delta_1 \Rightarrow \Delta_1' \qquad \vdash \Delta_2 \Rightarrow \Delta_2'}{\vdash \Delta_1; \Delta_2 \Rightarrow \Delta_1'; \Delta_2'}$$

(STRUCTDEF)
$$\frac{[f_1 : \phi_1, \ldots, f_n : \phi_n] \vdash \phi_i \qquad \phi_i \neq \texttt{private<}f\texttt{>}\ u}{\vdash t : (f_1 : \phi_1, \ldots, f_n : \phi_n) \Rightarrow t : (f_1 : \phi_1, \ldots, f_n : \phi_n)}$$

(GLOBAL)
$$\frac{\Gamma_G \vdash m\texttt{<}y\texttt{>}\ t \qquad m \neq \texttt{private}}{\vdash x : m\texttt{<}y\texttt{>}\ t \Rightarrow x : m\texttt{<}y\texttt{>}\ t}$$

(THREAD)
$$\frac{\Gamma = \Gamma_G[x_1 : \tau_1, \ldots, x_n : \tau_n] \qquad \Gamma \vdash \tau_i \qquad \Gamma \vdash s \Rightarrow s'}{\vdash f : \{x_1 : \tau_1, \ldots, x_n : \tau_n; s\} \Rightarrow f : \{x_1 : \tau_1, \ldots, x_n : \tau_n; s'\}}$$

$\Gamma \vdash \phi$	Type ϕ is valid in environment Γ.

(SAME)
$$\frac{t \in T}{\Gamma \vdash t}$$

(DEPENDENT)
$$\frac{\Gamma(x) = m\texttt{<}x\texttt{>}\ u \qquad t \in T}{\Gamma \vdash m\texttt{<}x\texttt{>}\ t}$$

$\Gamma \vdash \ell : \tau, \omega$	In environment Γ, ℓ is a well-typed lvalue with type τ assuming ω holds.

(NAME)
$$\frac{\Gamma(x) = m\texttt{<}y\texttt{>}\ t}{\Gamma \vdash x : m\texttt{<}y\texttt{>}\ t, \epsilon}$$

(SAME FIELD)
$$\frac{\Gamma(x) = m\texttt{<}y\texttt{>}\ t \qquad t(f) = t'}{\Gamma \vdash x.f : m\texttt{<}y\texttt{>}\ t', m(x)}$$

(DEPENDENT FIELD)
$$\frac{\Gamma(x) = m\texttt{<}x\texttt{>}\ t \qquad t(f) = m'\texttt{<}g\texttt{>}\ t'}{\Gamma \vdash x.f : m'\texttt{<}x.g\texttt{>}\ t', m(x)}$$

Figure 5.3: Typing judgments for definitions, types, and lvals.

$\Gamma \vdash s \Rightarrow s'$	In environment Γ statement s compiles to s', which is identical to s except for added runtime checks.

(NEW)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x := \text{new} \Rightarrow x := \text{new when } \mathrm{D}(\Gamma, x)}$$

(GROUP CAST)
$$\frac{x \neq y \qquad \Gamma(x) = \tau \qquad \Gamma(y) = m'\text{<}y\text{>}\ t}{\Gamma \vdash x := \text{gcast } y \Rightarrow x := \text{gcast } y \text{ when } \mathit{oneref}(y), \mathrm{D}(\Gamma, x), \mathrm{D}(\Gamma, y)}$$

(SHARING CAST)
$$\frac{x \neq y \qquad \Gamma(x) = m\text{<}w\text{>}\ t \qquad \Gamma(y) = m'\text{<}z\text{>}\ t'}{\Gamma \vdash x := \text{scast } y \Rightarrow x := \text{scast } y \text{ when } \mathit{oneref}(y), \mathrm{NF}(y), \mathrm{D}(\Gamma, x), \mathrm{D}(\Gamma, y)}$$

(READ)
$$\frac{\Gamma(x) = m\text{<}x'\text{>}\ t \qquad \Gamma \vdash \ell : m\text{<}\ell'\text{>}\ t, \omega}{\Gamma \vdash x := \ell \Rightarrow x := \ell \text{ when } \mathrm{R}(\omega), x'[\ell/x] = \ell', \mathrm{D}(\Gamma, x)}$$

(WRITE)
$$\frac{\Gamma \vdash x.f : m\text{<}\ell\text{>}\ t, \omega \qquad \Gamma(y) = m\text{<}y'\text{>}\ t}{\Gamma \vdash x.f := y \Rightarrow x.f := y \text{ when } \mathrm{W}(\omega), \ell[y/x.f] = y', \mathrm{D}(\Gamma, x.f)}$$

Runtime checks for assignments

$$\mathrm{NF}(x) = \{y.f = \text{null} | t'(f) = t''\}$$
$$\mathrm{D}(\Gamma, x) = \{y = \text{null} \mid x \neq y \wedge \Gamma(y) = m\text{<}x\text{>}\ t\}$$
$$\mathrm{D}(\Gamma, x.f) = \{x.g = \text{null} \mid \Gamma(x) = m\text{<}y\text{>}\ t \wedge f \neq g \wedge t(g) = m\text{<}f\text{>}\ t\}$$
$$\mathrm{R}(m(x)) = \begin{cases} \mathit{chkread}(x) \textbf{ if } m = \texttt{dynamic}; \\ m(x) \textbf{ otherwise} \end{cases} \qquad \mathrm{W}(m(x)) = \begin{cases} \mathit{chkwrite}(x) \textbf{ if } m = \texttt{dynamic}; \\ m(x) \textbf{ otherwise} \end{cases}$$

Figure 5.4: Typing judgments for statements. We omit the straightforward rules for non-assignment statements.

$$
\begin{array}{rcl}
M, id : \text{lock } x & \xrightarrow{s} & M[M_v(id.x) \xrightarrow{L} id] \textbf{ if } M_L(M_v(id.x)) = 0 \\
M, id : \text{unlock } x & \xrightarrow{s} & M[M_v(id.x) \xrightarrow{L} 0] \textbf{ if } M_L(M_v(id.x)) = id \\
M, id : x := \text{new} & \xrightarrow{s} & extend(M[id.x \xrightarrow{v} a], id, a, rtype(M[id.x \xrightarrow{v} a], id, M_\rho(id.x))) \\
 & & \textbf{where } a = \max(\text{dom}(M)) + 1 \\
M, id : x := \text{gcast } y & \xrightarrow{s} & gcast(M', id, c, rtype(M', id, M_\rho(id.x))) \\
 & & \textbf{where } c = M_v(id.y), M' = M[id.x \xrightarrow{v} c, id.y \xrightarrow{v} 0] \\
M, id : x := \text{scast } y & \xrightarrow{s} & scast(M', id, c, rtype(M', id, M_\rho(id.x))) \\
 & & \textbf{where } c = M_v(id.y), M' = M[id.x \xrightarrow{v} c, id.y \xrightarrow{v} 0] \\
M, id : \ell := \ell' & \xrightarrow{s} & M[lval(\ell) \xrightarrow{v} M_v(lval(\ell'))]
\end{array}
$$

Figure 5.5: Small steps in the operational semantics.

5.1.2 Operational Semantics

The parallel operational semantics of Figure 5.7 models the fine-grained interleaving of threads in a shared memory setting. The shared memory $M : \mathbb{N}^+ \to \rho \times \mathbb{N} \times \mathbb{N} \times (name \to \mathbb{N}) \to P(\mathbb{N}) \to P(\mathbb{N})$ maps a cell's address to its runtime type, owner, locker, value, and reader and writer sets. The runtime types ρ are of the form $m\text{<}a\text{>}\ t$, where a is the address of the cell's group leader. The owner is the thread identifier (an integer) that owns the cell if its sharing mode is `private`. The locker is the thread identifier that currently holds a lock on the cell, or 0 if the cell is unlocked. The value is a map from field names to cell addresses. Finally, the reader and writer sets are empty for objects not in the `dynamic` mode, and otherwise contain the identifiers of the threads that have read and written the cell. We use the notation $M_\rho(a)$, $M_o(a)$, $M_L(a)$, $M_v(a)$, and $M_R(a)$ and $M_W(a)$ to denote respectively the type, owner, locker, value, and reader and writer sets of cell a, while $M[a \xrightarrow{\rho} \rho]$, $M[a \xrightarrow{o} n]$, $M[a \xrightarrow{L} n]$, $M[a \xrightarrow{v} v]$, and $M[a \xrightarrow{R} n]$ and $M[a \xrightarrow{W} n]$ represent the corresponding updates to cell a. For convenience, we also write $M_v(a.f)$ and $M_\rho(a.f)$ to return respectively the value and type of field f of cell a, and $M[a.f \xrightarrow{v} b]$ to update field f of cell a. Note that $M_\rho(a.f)$ returns the static type τ of field f, not the runtime type of cell $M_v(a.f)$.

Thread identifiers are simply the address of a memory cell whose value is a map of the thread's variable names to their values. States of the operational semantics are of the form

$$M, (id_1, s_1), \ldots, (id_n, s_n)$$

representing a computation whose memory is M and which has n threads, each identified by address id_i and about to execute s_i. To handle thread creation, each thread definition f is represented by a *prototype thread* of the form $(id_f, \textbf{wait}; s_f)$ where s_f is the body of thread f and $M(id_f)$ is a memory cell with the initial state of f's variables: $M_v(id_f.x)$ is null for all local variables of f and a pointer to the preallocated structure instance o_x for global variable x.

Transitions between states are defined with a number of helper judgments and functions:

- $M, id : s \xrightarrow{s} M'$: execution of simple statement s by thread id updates the memory M to M'.

$$\frac{M_v(lval(\ell)) = M_v(lval(\ell'))}{M, id \models \ell = \ell' \to M} \qquad \frac{M_v(lval(\ell)) = 0}{M, id \models \ell = \text{null} \to M}$$

$$\frac{}{M, id \models \texttt{private}(x) \to M} \qquad \frac{M_L(M_v(id.x)) = id}{M, id \models \texttt{locked}(x) \to M} \qquad \frac{|\{b.f | M_v(b.f) = M_v(id.x)\}| = 1}{M, id \models oneref(x) \to M}$$

$$\frac{M_W(id.x) - \{id\} = \emptyset}{M, id \models chkread(x) \to M[id.x \xrightarrow{R} M_R(id.x) \cup \{id\}]}$$

$$\frac{M_W(id.x) - \{id\} = \emptyset \qquad M_R(id.x) - \{id\} = \emptyset}{M, id \models chkwrite(x) \to M[id.x \xrightarrow{W} M_W(id.x) \cup \{id\}]}$$

Figure 5.6: Runtime checks.

(SIMPLE STATEMENT)

$$\frac{M, id : s_1 \xrightarrow{s} M'}{\begin{array}{c} M, \{(id, s_1; s_2)\} \oplus S \;\to \\ M', \{(id, s_2)\} \cup S \end{array}}$$

(RUNTIME CHECK)

$$\frac{M, id \models \omega_1 \to M'}{\begin{array}{c} M, \{(id, \ell := e \text{ when } \omega_1, \omega_2 \ldots, \omega_n; s)\} \oplus S \;\to \\ M', \{(id, \ell := e \text{ when } \omega_2 \ldots, \omega_n; s)\} \cup S \end{array}}$$

(THREAD CREATION)

$$\frac{\begin{array}{c} id' = \max(\text{dom}(M)) + 1 \\ M' = extend(M, id', id', M_\rho(id_f)) \qquad M'' = M'[id' \xrightarrow{v} M_v(id_f)] \end{array}}{\begin{array}{c} M, \{(id, \text{spawn } f(); s), (id_f, \textbf{wait}; s_f)\} \oplus S \;\to \\ M'', \{(id', s_f; \textbf{done}), (id, s), (id_f, \textbf{wait}; s_f)\} \cup S \end{array}}$$

(THREAD DESTRUCTION)

$$\frac{M' = M \backslash id}{\begin{array}{c} M, \{(id, \textbf{done})\} \oplus S \;\to \\ M', S \end{array}}$$

Figure 5.7: Operational semantics.

$$lval(M, id, x) = id.x \quad lval(M, id, x.f) = M_v(id.x).f \textbf{ if } M_v(id.x) \neq 0$$

$$rtype(M, id, m\text{<}x\text{>}\ t) = m\text{<}M_v(id.x)\text{>}\ t$$
$$extend(M, id, a, \rho) = M[a \to (\rho, id, 0, \lambda f.0, \emptyset, \emptyset)]$$

$$gcast(M, id, c, \rho) = M' \textbf{ where } \begin{cases} M'_\rho(a) = \rho \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_\rho(a) \textbf{ otherwise} \\ M'_o(a) = id \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_o(a) \textbf{ otherwise} \\ M'_v(a) = M_v(a), M'_L(a) = M_L(a) \\ M'_R(a) = \emptyset \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_R(a) \textbf{ otherwise} \\ M'_W(a) = \emptyset \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_W(a) \textbf{ otherwise} \end{cases}$$

$$scast(M, id, c, \rho) = M' \textbf{ where } \begin{cases} M'_\rho(a) = \rho \textbf{ if } a = c, M_\rho(a) \textbf{ otherwise} \\ M'_o(a) = id \textbf{ if } a = c, M_o(a) \textbf{ otherwise} \\ M'_v(a) = M_v(a), M'_L(a) = M_L(a) \\ M'_R(a) = \emptyset \textbf{ if } a = c, M_R(a) \textbf{ otherwise} \\ M'_W(a) = \emptyset \textbf{ if } a = c, M_W(a) \textbf{ otherwise} \end{cases}$$

Figure 5.8: Auxiliary functions for the operational semantics.

- $M, id \models \omega \to M'$ holds if runtime check ω is valid, and updates the memory M to M'.
- $lval(M, id, \ell)$ converts a static lvalue ℓ of thread id to a runtime lvalue of the form $a.f$ (field f of cell a), and is undefined if ℓ requires dereferencing a null pointer.
- $rtype(M, id, \tau)$ converts a static type τ of thread id to the corresponding runtime type ρ.
- $extend(M, id, \rho)$ represents allocation of a cell of runtime type ρ by thread id.
- $gcast(M, id, c, \rho)$ performs a group cast to ρ in thread id of the group whose leader is $M(c)$.
- $scast(M, id, c, \rho)$ performs a single-object sharing cast to ρ in thread id of the single object at cell c.

The rules are mostly straightforward, but a few points are worth noting. We use $\oplus$ to denote disjoint union ($A = B \oplus C \Rightarrow B \cap C = \emptyset$). Individual runtime checks are executed atomically, but involve at most one lvalue of the form $x.f$. Other threads may execute between runtime checks and between the checks and the assignment they protect. The *oneref*(x) check is computed by heap inspection, but we implement it in practice using reference counting. We could exclude non-dependent fields from *oneref*(x) check, which would correspond to not reference-counting **same** pointers.

A group cast of a variable y whose value is null should have no effect. To ensure this, *gcast* never updates cells whose type is $m\text{<}0\text{>}\ t$. Effectively, group 0 corresponds to our `nogroup` C-level

annotation: once in group 0, you can no longer be subject to a group cast. Our formalism does allows casts to group 0, and allocations in group 0. Finally, and most importantly, the type and owner fields of M are not required in an actual implementation, thus *gcast* has no runtime cost outside of the null checks for the dependent type, and the reference counting for the sharing mode change.

5.1.3 Soundness

In Appendix A we prove the following theorems, which show that type safety is preserved and that the program respects the behavior specified by the sharing mode declarations.

Theorem 1 *Soundness.* Let P be a program with $\vdash P \Rightarrow P'$ and M_0, S_0 be the initial memory and threads for program P'. Let $M, \{(id_1, s_1), \ldots, (id_n, s_n)\}$ be a state reachable in i steps from M_0, S_0. Then types and owners are consistent in M, and all statements s_i type check.

Theorem 2 *Safety of* `private` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{private<}b\texttt{>}\ t$ then $M_o(a) = id$.

Theorem 3 *Safety of* `locked` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{locked<}b\texttt{>}\ t$ then $M_L(a) = id$.

Theorem 4 *Safety of* `dynamic` *access.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id accesses cell a with type $M_\rho(a) = \texttt{dynamic<}b\texttt{>}\ t$ then $id \in M_W(a) \Rightarrow M_R(a) \subseteq M_W(a) = \{id\}$.

Chapter 6

Implementing SharC

> There is a great satisfaction in building good tools for other people to use.
>
> *Disturbing the Universe*
> FREEMAN DYSON

We have implemented SharC in about 5500 lines of OCaml using the CIL [71] library, along with a runtime library consisting of about 1200 lines of C. We were able to apply this implementation to several large C programs. The input to SharC is a partially annotated C program, which is assumed to be type- and memory-safe if it were not multi-threaded. SharC first infers the missing annotations (Section 6.1). SharC then type-checks and instruments with runtime-checks the now-complete program. This augmented code is then passed to a normal C compiler and linked with SharC's runtime library.

At runtime, SharC verifies correct use of `dynamic` and `locked` locations, and checks that sharing and group casts are only applied to objects to which there is only one reference (Section 6.2).

6.1 Thread-Escape Analysis

It is sound to complete all unannotated types with the `dynamic` mode. The purpose of SharC's thread-escape analysis is to determine which unannotated types may safely be completed with the `private` mode. In particular, the unannotated types of objects that do not thread-escape are inferred to be in the `private` mode by SharC.

Before this step, however, to reduce the annotation burden, SharC will infer other sharing modes by following these simple rules:

- A field or variable used in a `locked` qualifier must be `readonly`, to preserve soundness.
- Type definitions can specify that they are inherently racy. This is used, e.g., for pthread's `mutex` and `cond` types.

- SharC provides a simple form of qualifier polymorphism for structs. If the outermost qualifier on a structure field is not specified, it is inferred to be the same as the qualifier on the *instance* of the structure. This is sound, since structure fields occupy the same memory that is described by the instance. As a consequence, SharC does not allow the outermost annotation of a field to be `private`: within a `private` struct, such a field is already private, while accesses to a `private` field within a non-`private` struct would be unsound.

- Outside of structure definitions, if the target type of a pointer is unannotated, then it is assumed to be the type of the pointer. For instance (`int * dynamic`) becomes (`int dynamic * dynamic`), but (`int dynamic * private`) remains as is. This inference is also used inside of structure type definitions unless an instance of the type may be subject to a group cast. In this case unannotated pointer target types default to `same`.

- An array is treated like a single object of the array's base type.

We have found that these rules expedite the application of SharC to real C programs.

After these rules have been applied, if it is enabled, the thread-escape analysis makes all remaining unannotated types either `private` or `dynamic`. Because accesses to `dynamic` objects are checked at runtime to detect data races, SharC attempts to minimize the number of objects inferred to be `dynamic`. First, we describe how the `dynamic` qualifier flows for assignments and function calls. Second, we describe how the analysis is seeded by a set objects that are inherently shared among threads. Taken together, this is sufficient to identify all the potentially shared objects that need the `dynamic` qualifier.

For assignments, we follow CQual's[37] flow-insensitive type qualifier rules, with changes to account for qualifier polymorphism in structures. To avoid overaggressive propagation of the `dynamic` qualifier, we only infer that it flows from formals to actuals in the following case: if a formal is stored in a `dynamic` location, or has a `dynamic` location stored in it, then the `dynamic` qualifier will flow to the actual at the call site. This is equivalent to adding a second kind of `dynamic` qualifier, which we internally refer to as *dynamic_in*, which accepts both `private` and `dynamic` objects. Users of our system never see or write this qualifier.

Next, we must find the shared objects with which to seed the analysis. First, we observe that for an object to be shared, it must be read or written in a function spawned as a thread. The locations available to a function spawned as a thread are the following:

- locals — These can only be shared if their addresses are written into another shared location, so locals are not seeds.

- formals — The argument to the thread function is an object passed by another thread, so is inherently shared and seeds the analysis.

- globals — All globals touched by thread functions might be shared, and so are seeds for the analysis.

It is an error if any of these inherently shared objects have been annotated by the programmer as `private`.

In order to identify all globals that might be touched by threads, we construct control flow graphs rooted at the functions that are spawned as threads. We handle function pointers by assuming that they may alias any function in the program of the appropriate type. This is sound under our type and memory safety assumption.

6.2 Runtime Checks

At runtime, SharC tracks held locks, reference counts, and reader–writer sets for `dynamic` and `barrier` locations. This information is then inspected in a straightforward way by the various runtime checks.

6.2.1 Tracking Reader and Writer Sets

The sets of threads reading and writing a location in memory is tracked for objects in the `dynamic` mode. The goal of this dynamic analysis is to enforce that `dynamic` objects are either readonly, or private to a thread. For every 16 bytes of memory SharC maintains an extra n bytes that record how each thread has accessed those 16 bytes. We can support up to $2^{8n-1} - 1$ threads when n extra bytes are used for record keeping. For the applications we have evaluated with SharC, setting $n = 1$ has been sufficient. Accesses and updates to the bits are made atomic, as required by the *chkread* and *chkwrite* checks described in Chapter 5, through use of the `cmpxchg` instruction available on the x86 architecture. These extra bytes encode the reader and writer sets as follows. If the low (0-th) bit is set, this indicates that a single thread is reading and writing the location, and the high $n - 1$ bits store the owners thread id. Otherwise, when the low bit is unset, the high $n - 1$ bits indicate the number of threads reading a location. This technique also requires each thread to keep track of the locations that it is reading in a bit array with each bit corresponding to a 16 byte chunk of memory. This is necessary to safely transition a location from being read by one thread to being owned by that thread.

When heap memory is deallocated with `free()`, it is no longer considered to be accessed by any thread, and all of its bits are cleared. When a thread ends, the bits recording its accesses are cleared: SharC does not consider it a race for two threads to access the same location if their execution does not overlap. The clearing operation is made efficient by logging the addresses of all of a thread's reads and writes to `dynamic` objects on its first accesses to those addresses.

6.2.2 Tracking Held Locks

When a lock is acquired, the address of the lock is stored in a thread private log. When a thread accesses an object in the locked sharing mode, a runtime check is added that ensures the required lock is in the log. When the lock is released, the address of the lock is removed from the log.

6.2.3 Checking Sharing Casts

The procedure for checking a sharing cast is given in Figure 6.1. When a programmer adds an explicit sharing cast, e.g. `x = SCAST(t, y)`, SharC transforms it into `x = scast(y, &y)` after

```
1 void *scast(void *src, void **slot) {
2   *slot = NULL;
3   if (refcount(src) > 1)
4     error();
5   return src;
6 }
```

Figure 6.1: The procedure for checking a sharing cast.

determining that the types are appropriate for a cast. The address of the reference is needed so that the reference can be nulled out. Then, if there is more than one reference to the object being casted, an error is signaled. Finally, the reference is returned. In the example above, if `y` were still live after the cast, SharC would issue a warning. This warning lets the programmer know that the reference will be null after the cast. The procedure for determining reference counts is described in section 6.2.4.

6.2.4 Maintaining Reference Counts

SharC extends the Heapsafe [43] framework with reference counting for multi-threaded C code. Applying Heapsafe without modification to SharC would imply atomically updating reference counts for all pointer writes. The resulting overhead is unacceptable on current hardware, even on x86 family processors that support atomic increment and decrement instructions. To reduce this overhead, SharC can optionally perform a straightforward whole-program, flow-insensitive, type-qualifier-like analysis to detect which locations might be subject to a sharing cast. Only pointers to these locations need reference count updates. To further reduce the overhead, we adapted Levanoni and Petrank's high performance concurrent reference counting algorithm [62] (designed for garbage collection) for use with SharC.

In Levanoni and Petrank's algorithm, there are a number of threads mutating data(i.e. *mutators*), and a distinguished thread that performs garbage collection(i.e. the *collector*). Each mutator thread keeps a private, unsynchronized log of the reference updates it performs. The log contains a record for each update of which reference was updated along with the old value that was overwritten by the update. To keep the log size manageable, an entry is only added the first time a reference is updated. This is accomplished by keeping a dirty bit for each reference, which is set the first time the reference is updated, and occasionally cleared by the collector thread.

When a reference count is needed, the collector thread stops the mutator threads, copies their logs before clearing them, resets the dirty bits, and starts the mutator threads running again. The collector thread then processes the logs as follows. First, the reference counts for the overwritten values are decremented. Next, the reference counts for the new values are incremented, but only when the dirty bit for the reference cell has not been set again since the mutator threads were restarted. If the dirty bit has been set again, the reference count for the overwritten value in the currently live logs is incremented—being in the new logs, it will be decremented when the new logs are processed. We note here that it is safe to temporarily overestimate the reference counts. Levanoni and Petrank also have another algorithm that stops threads one by one, rather than all at once, but it is more complicated to implement.

We have adapted the simpler algorithm in the following ways. First, in our adaptation, there is no need for a dedicated collector thread. When a thread needs a reference count, it simply performs all the tasks of the collector thread listed above. However, only one thread at a time may act as the collector thread. Second, there is no need to stop all threads while the collector thread copies logs. Rather, in our implementation there are two sets of logs, and two arrays of dirty bits. Instead of copying logs, the collector thread arranges (through a simple lock-free algorithm) for each thread to use the other set of logs and dirty bits, and waits for any pending updates to complete. At this point the collector thread can proceed as before.

Levanoni and Petrank also describe how to avoid the problems that arise when the target architecture does not have a sequentially consistent memory model. We describe in detail our algorithm and how these problems are addressed by our implementation in Appendix C.

6.3 Groups

We reduced the additional annotation burden presented by the addition of groups by automatically applying common default annotations to unannotated objects. First, the default group annotation on pointer types that need one is `nogroup`. This way, the programmer must only annotate objects needing a group annotation if they may eventually be involved in a group cast. Secondly, the default annotation on structure fields in objects that may be subject to a group cast is `same`. This prevents SharC from forbidding casts of these structures.

Support for groups requires little additional dynamic checking, i.e. reference counting, and checking for the `locked` and `dynamic` sharing modes. As mentioned above, our `group()` type is dependent, and as one might expect, some of the type checking must be done at runtime. In our case, this entails inserting a small number of checks to make sure that certain pointers are null. In practice the number of checks is small, and our time-measurement primitives are not precise enough to measure their affect on the performance of our benchmarks.

6.4 The C Library

In applying SharC to real C programs, it is necessary to handle some features of the C language, and the C Library. In particular, we require pointer arguments to variable argument functions to be `private`. This caused no problems when SharC was applied to the benchmarks in Chapter 10. Further, we stipulate that C Library calls require pointer arguments be `private`. However, SharC also supports trusted annotations that summarize the read/write behavior of library calls. When the read/write behavior of a library call is summarized for an argument, the call may accept an actual argument in any sharing mode except for `locked`. In particular, for a `dynamic` actual, the read/write summary tells how to update the reader/writer sets for the object, and a `readonly` actual can be safely passed when there is a read summary.

6.5 Failure Modes

When a violation of the declared sharing strategy occurs at runtime, our implementation provides two different options for reporting the failure to the programmer or user. Each of these reporting options is useful in different situations. In one option, SharC logs the runtime error to a file or the console, and continues. In the other, a runtime error causes the program to halt with an error message. The first option is useful in the process of adding SharC's annotations. The error messages provided in this situation are a good indication of how objects are shared among thread in the program. After it seems that a program has been annotated correctly, the second option is useful for indicating the now rare case that the program is annotated incorrectly, or when there is a real sharing bug in the program.

6.6 Limitations

SharC has a few limitations. First, false race reports may result from false sharing, and from the use of custom memory allocators. Since we track races at a 16-byte granularity, races may be reported for two separate objects that are close together, but used in a non-racy way. To alleviate this problem, SharC ensures that `malloc` allocates objects on a 16-byte boundary. If a program's custom memory allocator transfers unused memory between threads, or does not allocate on 16-byte boundaries, SharC may incorrectly report races. In the future, we will provide support for making SharC understand custom allocators.

Second, our analysis is dynamic, so it will only detect sharing strategy violations over a limited number of paths, and only for thread schedules that actually occur on a concrete run of the program. The advantage of the dynamic analysis is, of course, that errant program behavior will be detected when it occurs, rather than at some later time.

Finally, it is unlikely that the sharing modes presented here are sufficient to describe the sharing strategies found in every concurrent C program. For example, by chance, a fairly common initialization pattern did not appear in our benchmarks. In this pattern an object is protected by a lock when in some "bad" state, but becomes readonly when in a "good" state. Additionally, not all concurrent programs are based on threads. For example, event-based programs, SPMD programs, and OpenMP-based programs use different concurrency models. Since the concurrency model is different, even though C syntax is used, the underlying semantics of the language are different, and would require possibly substantially different analyses to the ones presented here. However, we believe that expressing sharing rules as type-qualifiers could apply in general to these various models.

Part III

Shelters

Chapter 7

Design of Shelters

You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

Linux 1.3.53 CodingStyle Documentation
LINUS TORVALDS

The `sheltered_by(s)` sharing mode of SharC is a mode for objects for which concurrent access is mediated by the shelter `s`. Objects protected by a shelter may only be accessed within a programmer supplied `atomic` section. This chapter describes the design of shelter-based atomic sections, and gives an example of their use.

7.1 Design of Shelters

In this section we describe our extensions to C, and explain how these extensions are translated into calls into our system's runtime. We also present a small example that demonstrates many of the features of our system.

In addition to atomic sections, our system adds annotations to C for describing what objects require protection from concurrent access, and what the sheltering needs of functions are when called from inside of an atomic section. The resulting extensions to the C language are:

- `atomic {...}` — Atomic statements indicate that the effects of the statements on shared objects in the indicated block should not become visible to other threads until execution exits the block.
- `shelter_t` — Shelters are first-class objects in the language, and can be declared as variables, structure fields, or arguments to functions.
- `sheltered_by(s)` — This is an annotation on types indicating that an object of the annotated type can only be accessed in an atomic section (or a function called within an atomic section), and that concurrent access is mediated by the shelter `s`.

- `needs_shelters(s1,s2,...)` — This is an annotation on function types that contains a list of the C expressions of type `shelter_t` in terms of formal parameters and global variables. The list must contain expressions for shelters that protect objects read and written by the function. The list may be empty if no sheltered expressions are accessed. Only functions called from within atomic sections require an annotation.

Our system issues warnings and errors at compile-time as appropriate when required `needs_shelters` annotations are missing from functions, and when objects with a `sheltered_by` annotation are accessed outside of an atomic section or annotated function. The annotation burden imposed by our system, and a comparison to similar systems is investigated in Chapter 11.

Our system takes as input a program written using these extensions to C, and outputs a program that makes calls into a runtime implementing shelters-based atomic sections, which we then pass to an off-the-shelf C compiler for compilation and linking. This translation takes place in two steps. First, we perform a backwards dataflow analysis over atomic sections in order to collect the shelters that protect objects read and written in the atomic section. Where the analysis is imprecise, we use coarser-grained shelters as described in Section 7.1.2 below. In the second step, we use the results of the dataflow analysis to instrument the program with calls into our system's runtime.

Since the backwards analysis is straightforward, we only describe the runtime calls that are inserted by the analysis and instrumentation:

- `shelter_register(s,...)` — This call is added at the beginning of an atomic section. The arguments are the shelters that were found by the backwards analysis. In one atomic action, the call acquires one globally unique, increasing sequence number, and adds the calling thread to queues for each of the shelters in the argument list. Our implementation uses lock-free algorithms for acquiring the sequence number and adding the thread to the shelters' queues.

- `shelter_wait(s)` — This call is added when an object of a type annotated with `sheltered_by(s)` is accessed. The call causes the calling thread to wait until it is the thread with the smallest sequence number on `s`'s queue.

- `shelter_release_all()` — This call is added at the end of an atomic section. It causes the calling thread to remove itself from the queues of the shelters for which it had registered.

We support nested atomic sections by ignoring nested calls to `shelter_register` and `shelter_release_all`.

7.1.1 Example

We now show how shelter-based atomic sections can avoid the pitfalls of explicit locking while achieving the convenience of atomic sections for the example of the atomic transfer of funds between two bank accounts.

Consider the type and function declarations in Figure 7.1. The `account_t` structure type contains a `balance` field, and an `id` field. The structure also contains a `shelter_t` field `s` for

```
1  typdef struct {
2     int sheltered_by(s) id;
3     float sheltere_by(s) balance;
4     shelter_t s;
5  } account_t;
6
7  needs_shelters(a->s)
8  void deposit(account_t *a, float d) {
9    a->balance += d;
10 }
11
12 needs_shelters(a->s)
13 void withdraw(account_t *a, float w) {
14   a->balance -= w;
15 }
16
17 needs_shelters(to->s, from->s)
18 void transfer(account_t *to, account_t *from,
19               float a) {
20   atomic {
21     withdraw(from, a);
22     deposit(to, a);
23   }
24 }
```

Figure 7.1: Code using atomic sections for the atomic transfer of funds between two accounts.

```
 1 void deposit(account_t *a, float d) {
 2   shelter_wait(a->s);
 3   a->balance += d;
 4 }
 5
 6 void withdraw(account_t *a, float w) {
 7   shelter_wait(a->s);
 8   a->balance -= w;
 9 }
10
11 void transfer(account_t *to, account_t *from,
12               float a) {
13   shelter_register(to->s, from->s);
14   withdraw(from, a);
15   deposit(to, a);
16   shelter_release_all();
17 }
```

Figure 7.2: The results of the transformation of code with atomic sections and shelter annotations to code with shelter registrations, waits, and releases.

protecting the `balance` and `id` fields, as indicated by the `sheltered_by(s)` annotations. The `deposit` and `withdraw` functions adjust the balance of an account. They must be annotated as `needs_shelter(a->s)` because they access fields of a that have been annotated as `sheltered_by(s)`. In the `transfer` function an `atomic` block indicates that the withdraw from the `from` account and the deposit to the `to` account happen atomically. Additionally, the `transfer` function must be annotated with `needs_shelters(to->s,from->s)` if it is ever going to be called from within an atomic section, because it calls functions that access data protected by those shelters.

Figure 7.2 shows the results of transformation of the program with atomic sections and annotations to code that registers for shelters, waits on them before accessing the protected objects, and releases them when finished. The beginning of the atomic section in the `transfer` function is translated into the `shelter_register()` call on Line 13, which atomically acquires a unique sequence number and adds the calling thread to the queues for shelters `to->s`, and `from->s`. These shelters are collected by the backwards analysis from the annotations on the `deposit` and `withdraw` functions.

Where the program accesses objects with types annotated with `sheltered_by(s)`, calls to `shelter_wait(s)` are added that cause the calling thread to wait until it is the thread with the smallest sequence number on the queue for `s`. These calls appear for the accesses to the `balance` field of account structures on Lines 2 and 7 of Figure 7.2.

Where the atomic section ends in Figure 7.1, we now have a call to `shelter_release_all()` on Line 16, which removes the calling thread from the queues of the shelters for which it had registered.

Now, consider that one thread calls `transfer(A, B)`, and a second thread calls `transfer(B, A)` at the same time. If the example in Figure 7.1 were written with explicit locking, care would

have to be taken when implementing or calling the `transfer` function in order to avoid deadlock. With shelter-based atomic sections, however, deadlock is avoided automatically because the two threads will have distinct sequence numbers, one smaller than the other. If two threads are registered for the same shelter `s`, and both arrive at a `shelter_wait(s)` call, the thread with the smaller sequence number proceeds while the thread with the larger sequence number must wait until the first thread calls `shelter_release_all()`.

7.1.2 Shelter Hierarchy

The backwards dataflow analysis cannot always statically infer precisely what shelters are needed on entry to an atomic section. We must therefore make use of a hierarchy of shelters, with coarser grained shelters used in response to imprecision in the analysis. Consider the following alternate version of the `transfer` function from the example:

```
void idTransfer(int toId, int fromId, float a) {
  atomic {
    account_t *to = accountLookup(toId);
    account_t *from = accountLookup(fromId);
    withdraw(from, a);
    deposit(to, a);
  }
}
```

In this example the function `accountLookup` takes the account ID, looks up the `account_t` structure in some data structure implementing a map, and returns a pointer to it. It might be preferable to write the function like this in case, for example, accounts may be deleted from the system. Depending on how the map data structure is implemented, it may not be possible for a static analysis to determine exactly what shelters are needed at the beginning of the atomic section. In this situation, our current implementation registers for a coarser grained shelter protecting *all* `account_t` structures, which subsumes the shelters in the individual `account_t` structures. We call these coarser-grained shelters *type-shelters*, and will refer to particular type-shelters as `T.s`, where `T` is the structure type name, and `s` is the shelter field, for example `account_t.s`. In our implementation they are only needed to subsume the shelters that are fields of structure types.

A static analysis having shape or ownership information may be able to obtain a finer-grained hierarchy. This is the approach suggested by McCloskey et al. [66], and implemented by Cherem et al. [20], and Hicks et al. [51], however these refinements are largely orthogonal to our contribution. Furthermore, our design goals include avoiding whole-program analysis or an intricate system of annotations, one of which approaches like these likely require.

In order to implement our hierarchy, when a thread registers for a shelter, it must also place itself on the queues of its ancestors in the hierarchy. Furthermore, when checking to see if it must wait for a shelter, a thread must also wait if a thread with a lower sequence number has registered directly for one of its ancestor shelters. For example, suppose thread T_1 is registered for a shelter protecting a particular `account_t` structure, `a->s`, and has sequence number 3. Further suppose that thread T_2 is in the atomic section in the alternate `transfer` implementation, and is registered

directly for the ancestor of a->s, account_t.s, with sequence number 2. Even if T_1 is the thread with the smallest sequence number on the queue of shelter a->s, it must wait because T_2 has a smaller sequence number and is on the queue for account_t.s, an ancestor of a->s. On the other hand, if T_2 had only been registered for the shelter for another account_t structure, say a'->s, then both threads would be able to proceed.

We also provide syntax for adding a shelter higher up in the hierarchy to the list of shelters in the needs_shelters annotation.

7.1.3 Condition Variables

Our implementation includes support for condition variables. That is, threads may send signals and wait on condition variables based on shared state that is protected by shelters. Shelter condition variables are much like traditional condition variables. They are declared like pthread condition variables, e.g. shelter_cond_t scv, and are signaled in the same way, e.g. shelter_cond_signal(scv). However, a conditional wait on shelter protected state is slightly different. We introduce the following construct:

```
shelter_cond_wait(scv,e) {
  stmts;
}
```

The meaning of this statement is as follows. The thread waits on the shelter condition variable scv while the condition, e, is false. If the thread is then signaled, and the condition is true, it executes the statements stmts atomically. This is accomplished by our analysis treating the shelter wait block as an atomic section and collecting the shelters necessary for protecting both the block and the condition e. Then, the above construct can be translated as follows.

```
shelter_register(S);
shelter_wait(S);
while(!e) {
  ll_shelter_release_and_wait(S,scv);
  shelter_register(S);
  shelter_wait(S);
}
stmts;
shelter_release_all();
```

Here, S is the set of shelters found by the analysis, and ll_shelter_release_and_wait(S,scv) atomically releases the shelters in S, and puts the thread to sleep waiting for a signal on scv. We leave as future work an extension to our system, like the one in Autolocker, that ensures that condition variables are signaled when appropriately specified state is updated.

7.1.4 Library Calls and Polymorphism

If a library call does not invoke any callbacks, it will not cause a thread to register for any shelters. Therefore, it is only necessary to know what objects such a library call will read and write in case any of these locations are protected by a shelter. We allow programmers to indicate this by providing annotations that summarize the read and write behavior of library calls, so that our implementation can automatically place the appropriate `shelter_wait` calls ahead of the library calls. Library calls invoking callbacks that access shelter protected state are not currently supported by our system.

In our current implementation we do not support type-qualifier polymorphism for the `sheltered_by(s)` annotations. This sort of feature has not been needed in the benchmarks we analyze in Chapter 11 due to the limited use of polymorphism in C programs. However, more modern languages may require increased support of polymorphism to support code-reuse, and other good software engineering practices. We leave support for polymorphism as future work.

7.1.5 Other Synchronization Strategies

It is not realistic to assume that all shared data will be protected by shelters and accessed within atomic sections. For instance, some shared data will be readonly and need no synchronization, while other data will be protected by other means: barrier synchronization, data obtained from work queues and worked on exclusively by a single thread, etc. Furthermore, external libraries may already use locks to protect their own data — converting these libraries to use shelters may not be desirable, practical or even possible.

Three issues must be considered in the resulting programs:

- The programmer must ensure that data is shared correctly and using consistent mechanisms. This can be accomplished using SharC or the work of Martin et al. [65], which uses dynamic ownership assertions to detect where such rules are violated.

- The programmer must ensure that the mix of synchronization mechanisms does not cause deadlock. Such deadlocks are possible even when each mechanism is used safely: for instance, holding a lock when calling a barrier-synchronization function can cause deadlock if it prevents another thread from reaching its own barrier call. Similarly, calling a barrier-synchronization function inside a shelter-based atomic section will likely cause deadlock. Holding any lock when a `shelter_wait(s)` call occurs (i.e. when a sheltered object is accessed within an atomic section) is also likely to cause deadlock. However, this still allows many safe uses of locks with our system. Locks can be freely used outside atomic sections. Functions using locks can be freely called from within an atomic section as long as they release all acquired locks before returning and make no calls to shelter-using functions — this should be true of the typical library that uses locks to protect its internal state.

- Atomicity can be violated in some cases. For instance, atomic sections that access both lock-protected and shelter-protected data are not guaranteed to be atomic. However atomic sections that access only thread-private, readonly and shelter-protected data do remain atomic.

Chapter 8

Formalizing Shelters

> It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.
>
> *On the Method of Theoretical Physics*
> ALBERT EINSTEIN

We show that atomic sections implemented using shelters do indeed provide atomicity. That is, execution with shelters is equivalent to a sequential interleaving of atomically executed sections. Formally, we prove that a valid execution trace of a shelter-based program is equivalent to the corresponding trace where the atomic sections are executed atomically. Our approach is fairly different to that used for Jade [78] as we are proving a different property (atomicity vs. equivalence to a sequential program).

A trace T, defined in Figure 8.1, captures the essential aspects of execution in an imperative language with shelters. A trace starts with the declaration (d_i) of the global integer variables that are used in the trace. Each variable v is protected by its own shelter v_σ and a global shelter s (possibly shared with other variables), mirroring the hierarchical shelters in our system. The

Trace	T	::=	$d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m)$
Statement	s	::=	$\text{atomic}_n(\sigma_1, \ldots, \sigma_m) \mid \text{endatomic}$
		$\mid$	$v := n \mid v := v_1 + v_2$
Declaration	d	::=	int v sheltered by s
Shelter	σ	::=	$v_\sigma \mid s$

Identifiers v, s Integers t, n, m

Figure 8.1: Traces of shelter-based programs.

$$\frac{\text{access}(A(t), v, \Sigma) \qquad \text{access}(A(t), v_1, \Sigma) \qquad \text{access}(A(t), v_2, \Sigma)}{M, A, a, \Sigma : (t, v := v_1 + v_2) \rightarrow M[v \rightarrow M(v_1) + M(v_2)], A, a, \Sigma}$$

$$\frac{A(t) = 0 \qquad n > a \qquad \Sigma' = \Sigma[\ldots, \sigma_i \rightarrow \Sigma(\sigma_i) \cup \{n\}, \ldots]}{M, A, a, \Sigma : (t, \text{atomic}_n(\sigma_1, \ldots, \sigma_m)) \rightarrow M, A[t \rightarrow n], n, \Sigma'}$$

$$\frac{\text{access}(A(t), v, \Sigma)}{M, A, a, \Sigma : (t, v := n) \rightarrow M[v \rightarrow n], A, a, \Sigma}$$

$$\frac{A(t) \neq 0 \qquad \text{dom}(\Sigma') = \text{dom}(\Sigma) \qquad \forall \sigma \in \text{dom}(\Sigma).\Sigma'(\sigma) = \Sigma(\sigma) \setminus \{A(t)\}}{M, A, a, \Sigma : (t, \text{endatomic}) \rightarrow M, A[t \rightarrow 0], a, \Sigma'}$$

$$\frac{M, A, a, \Sigma : (t_1, s_1) \rightarrow M', A', a', \Sigma'}{M, A, a, \Sigma : (t_1, s_1), \ldots, (t_m, s_m) \rightarrow M', A', a', \Sigma' : (t_2, s_2), \ldots, (t_m, s_m)}$$

$$\frac{\begin{array}{c}\forall v.M(v) = 0 \qquad \forall \sigma.\Sigma(\sigma) = \emptyset \\ \forall t.A(t) = 0 \qquad \forall t.A'(t) = 0 \qquad M, A, 0, \Sigma : (t_1, s_1), \ldots, (t_m, s_m) \xrightarrow{*} M', A', a', \Sigma' :\end{array}}{d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m) \rightarrow M'}$$

$$\begin{array}{rcl}\text{access}(n, v, \Sigma) & = & (n \in (\Sigma(v_\sigma) \cup \Sigma(G(v)))) \wedge \\ & & (\forall m \in \Sigma(v_\sigma) \cup \Sigma(G(v)).m \geq n)\end{array}$$

Figure 8.2: Trace Operational Semantics

trace itself is a sequential interleaving of statements from multiple threads, where each thread is identified by a distinct integer t. The statements, executed atomically, are either the start of an atomic statement requiring shelters $\sigma_1, \ldots, \sigma_n$ (atomic), the end of an atomic statement, or assignments of an integer or computed value to a variable v.[1] An atomic statement has a sequence number n which must be greater than all atomic sequence numbers found earlier in the trace.

Traces do not necessarily represent a valid execution of a shelter-based program. For instance,

$$\text{int x sheltered by f}, (0, \text{atomic}_{42}(\text{g})), (0, \text{x} = 2), (0, \text{endatomic})$$

accesses x without holding either its individual shelter x_σ or its global shelter f. In Figure 8.2 we give an operational semantics for traces that computes a trace's effects as long as the trace is valid. The state of the operational semantics is a four-tuple M, A, a, Σ where $M : id \rightarrow \mathbb{N}$ maps variables to their values, $A : \mathbb{N} \rightarrow \mathbb{N}$ maps threads to their current atomic statement, $a : \mathbb{N}$ is the sequence number of the last initiated atomic statement and $\Sigma : \sigma \rightarrow \mathcal{P}(\mathbb{N})$ maps shelters to the set of active atomic statements that have requested that shelter. Finally, we write $G(v)$ to represent the global shelter specified in v's declaration.

[1] The atomic execution of $v := v_1 + v_2$ is not essential to the proof and could easily be relaxed with the addition of per-thread variables to the trace.

The two assignment rules use the *access* function to check that thread t's current atomic statement $A(t)$ has either requested access to variable v's shelter v_σ or its global shelter $G(v)$, and that no atomic statement with an earlier sequence number currently has access to either of these two shelters. The rules then update the memory with the assignment's result. The atomic statement rule has three parts. First, the current thread must have ended any previous atomic statement ($A(t) = 0$). Second, the atomic statement's sequence number n must be greater than that of the previous (from any thread) atomic statement a. Finally, it computes Σ', the new state of shelter requests, by adding the atomic statement sequence number n to the requested shelter sets. Ending an atomic statement is symmetric: the thread must be executing an atomic statement ($A(t) \neq 0$), then it computes the new shelter state by removing the atomic statement's sequence number $A(t)$ from Σ and sets its atomic statement number to 0. The last two rules evaluate a complete trace from the initial state $M(v) = 0$ (all variables initialized to zero), $A(t) = 0$ and $a = 0$ (no active atomic statement), and $\Sigma(\sigma) = \emptyset$ (no shelters requested by any thread). Note that we also require that at the end of the trace all threads have ended their atomic statements ($A'(t) = 0$).

Definition 1 Trace $d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m)$ is *valid* with results M if $d_1, \ldots, d_n, (t_1, s_1), \ldots, (t_m, s_m) \rightarrow M$.

Definition 2 A trace is *serial* if it is of the form

$$
\begin{aligned}
&d_1, \ldots, d_n,\\
&(t^1, \text{atomic}_{n^1}(\ldots)), (t^1, v^1_1 := \ldots), \ldots, (t^1, v^1_{m^1} := \ldots), (t^1, \text{endatomic}),\\
&\ldots\\
&(t^k, \text{atomic}_{n^k}(\ldots)), (t^k, v^k_1 := \ldots), \ldots, (t^k, v^k_{m^k} := \ldots), (t^k, \text{endatomic})
\end{aligned}
$$

Such traces execute the body of an atomic statement without any interference by other threads, so are obviously atomic.

Definition 3 *ops (t, T)* is the subsequence of statements in trace T executed by thread t.

Definition 4 *atomicorder (T)* is the subsequence of atomic statements of T.

Theorem 5 *Atomicity.* For every valid trace T with results M there exists a serial trace T' with results M. Furthermore, $\text{atomicorder}(T) = \text{atomicorder}(T')$ and for every thread t, $\text{ops}(t, T) = \text{ops}(t, T')$.

Less formally, the execution of T is equivalent to the clearly atomic execution obtained by moving all the assignments in each atomic section so that they occur immediately after the atomic statement.

Proof: The proof proceeds by a step-wise transformation of T into a serial trace T'. The detailed proof appears in Appendix B

Chapter 9

Implementing Shelters

An algorithm must be seen to be believed.

The Art of Computer Programming
DONALD KNUTH

Our implementation of shelter-based atomic sections comprises a 2200 line extension to Ivy and SharC, and includes a runtime library written in about 2500 lines of C. We use a combination of lock-free algorithms and other optimizations to ensure that our implementation scales.

9.1 Shelter Registration and Waiting

In order to avoid threads being serialized in acquiring a unique sequence number and adding themselves to shelter queues, we use lock-free algorithms for the `shelter_register` and `shelter_wait` operations. Figures 9.1 and 9.2 give sketches for these functions. The intention is to illustrate the key features of the algorithms rather than to show a complete implementation.

In Figure 9.1, we first define two structures, one for the shelters themselves, and the other for the entries in the shelters' queues. The `shelter_t` type includes the shelter's queue, `Q`; indexes for the front and back of the queue, `front` and `back`; the shelter's level in the hierarchy, `level`, where a smaller number indicates that the shelter is higher in the hierarchy; and a pointer to the shelter's parent in the hierarchy, `parent`. The entries in the queue, with type `qentry_t`, hold pointers to the sequence numbers of the registered threads in the `thread` field, and the levels of the shelters that those threads have initially registered for in the `level` field.

Two global variables keep track of the global sequence number counter, `global_counter`, and the thread-private sequence number, `T`. We use 64-bit unsigned integers so that overflow is unlikely.

The goal of shelter registration is to atomically acquire a unique sequence number, and place the calling thread in the right place on the indicated shelters' queues. The thread must be added to all shelter queues atomically in order to avoid deadlock. The function works as follows. First the global sequence number is read (Line 17). Then the thread adds itself to the queues for the indicated shelter and its ancestors (Line 18). We omit the pseudo-code for the queue because

```
1  typedef struct {
2    qentry_t Q[SIZE];
3    int front, back, level;
4    shelter_t *parent;
5  } shelter_t;
6
7  typdef struct {
8    uint64_t *thread;
9    int level;
10 } qentry_t;
11
12 uint64_t global_counter;
13 __thread uint64_t T; // T is thread-private
14
15 void shelter_register(shelter_t *s) {
16 retry:
17   uint64_t old = global_counter;
18   addToQueues(s, &T);
19   T = old + 1;
20   if (CAS(&global_counter, old, T) != old) {
21     T = 0;
22     rmFromQueues(s, &T);
23     goto retry;
24   }
25 }
```

Figure 9.1: Psuedo-code for the shelter register function.

it is a standard lock-free queue implemented with a circular buffer. Next, the thread's sequence number is assigned to the next available sequence number (Line 19) before an attempt is made to increment the global sequence number counter with an atomic compare-and-swap (Line 20). If that fails, the thread zeroes out its sequence number (Line 21), and removes itself from the shelters' queues (Line 22) before retrying (Line 23).

The global sequence number is read before adding the thread to the shelter queues so that when the compare-and-swap succeeds, we are guaranteed that any unsorted-ness ahead of the thread in the queue will be only temporary. It is also the case that entries in the queue may temporarily be zero (that is, before assigning `T` after adding a thread to the queue, or after zeroing it when the compare-and-swap fails.) It is safe for the `shelter_wait` function to ignore these zeroed entries.

In Figure 9.2, the shelter wait function works by checking to see if it must wait for a shelter by inspecting its thread queue, and then making a recursive call to see if it must wait for any of that shelter's ancestors. It takes two arguments, the shelter currently being inspected, and the level in the hierarchy of the shelter that the thread initially called `shelter_wait` on. The call mentioned in Chapter 7, which only took the shelter argument, is a simple wrapper for this call.

First, the thread will look through the shelter's queue, either for its own entry or for an entry that means it must wait. This loop begins on Line 5. In the loop, the thread checks to see if it has

```
1  void shelter_wait(shelter_t *s, int level) {
2    int wait, front;
3    do {
4      wait = 0; front = s->front;
5      while (front != s->back) {
6        if (Q[front].thread == &T) break;
7        else if (*Q[front].thread &&
8                 level == s->level &&
9                 *Q[front].thread < T) {
10         wait = 1; break;
11       }
12       else if (*Q[front].thread &&
13                s->level < level &&
14                Q[front].level < level &&
15                *Q[front].thread < T) {
16         wait = 1; break;
17       }
18       front = (front + 1) % SIZE;
19     }
20   } while (wait);
21   if (s->parent) shelter_wait(s->parent, level);
22 }
```

Figure 9.2: Psuedo-code for the shelter wait function.

found its own entry (Line 6). Since a thread can never see its own entry out of order in the queue (shelter registration only completes when an in-order enqueue succeeds), when a thread finds its own entry, it does not need to wait. If a thread has not yet found itself, and if the the shelter currently being inspected is the one the thread initially called `shelter_wait` on, then the thread must wait if it finds an entry in the queue that has a smaller sequence number (Line 10). If the shelter currently being inspected is an ancestor of the shelter that the thread initially waited on, then the thread must only wait if it finds a queue entry for a thread that initially waited on a shelter higher in the hierarchy that *also* has a smaller sequence number (Line 16). On Line 21, the thread checks to see if it must wait for a parent shelter by making a recursive call to `shelter_wait`.

In this pseudo-code, when a thread discovers that it must wait, it simply retries the check. However, it is straightforward to implement other options, such as sleeping, or waiting on a traditional condition variable. In our experiments, we simply retry the check again immediately after an invocation of the scheduler, however in the future we intend to experiment with the futex [27] system calls provided by Linux.

9.2 Optimizations

In practice, a number of optimizations are required for our system to scale. In particular, the following optimizations were critical to achieving performance similar to, and in some cases,

better than explicit locking.

9.2.1 Important Optimizations

It is often the case that a shared object is only ever read in a atomic section. Our static analysis is able to determine when a shelter is only required by an atomic section for read access. This has been omitted from the runtime description above, however in these cases, a thread may indicate that it will only ever read objects protected by a shelter when registering for it. Then, when determining whether or not it should wait before reading a sheltered object, a thread must only have a sequence number smaller than threads on the shelter's queue that have registered for write access. This mechanism is essentially equivalent to explicit read/write locking, however our shelter-based atomic section implementation invokes it automatically wherever possible.

It is also often the case that atomic sections are used to protect access to objects for which there is not much contention. To take advantage of this, instead of adding itself to a shelter's queue, our runtime allows a thread to acquire a spinlock—or to increment a counter in the case of read-only access when there are no writers—during the registration phase in the case that there are no other threads on the shelter's queue.

Furthermore, not all programs will use the various levels of the shelter hierarchy, and those that do will not be using them at all times. Therefore, our runtime includes a mechanism to activate shelters higher up in the hierarchy only when they are needed—that is, when a thread attempts to register for them directly. When a shelter is inactive, threads registering for its children must simply read a flag that indicates inactivity, and check that its queue is empty to see that no further action is required. Threads also record for which inactive shelters they have registered. When a thread wishes to register directly for a shelter with children, it registers as usual, but during a `shelter_wait` call, it must also wait until there are no threads registered for a child shelter that make use of the inactivity of the parent shelter. This check is made by examining the inactive shelters that each thread has registered for. When there are no more such threads, we unset the inactive flag in the parent shelter, and proceed.[1]

As the number of threads and cores increases, contention for the global sequence number counter increases. To address this, when a compare-and-swap operation on the counter fails, after retrying immediately a small number of times, we use a binary exponential backoff algorithm [45]. In our experiments, this approach reduced by several orders of magnitude the number of compare-and-swap failures without compromising performance.

9.2.2 Other Optimizations

We also implemented a few obvious optimizations that we believe to be important in general, but for which we observed no advantage in our experiments.

In particular, from the results of our backwards analysis, we can deduce a few interesting facts. First, we find places where it is safe to release a shelter before the end of an atomic section.

[1] It may seem simpler for threads registering for a child shelter to simply acquire read-access to the inactive parent shelter by way of a counter, as in a reader-writer lock. However, in practice, contention for this counter can incur an unacceptably high overhead.

Similarly, we find places where it is safe to downgrade a thread from write access to read access. Finally, we find places where it is safe to give up a parent shelter, and to instead acquire one of its children. These optimizations were left enabled for the results we present in Chapter 11, but because the atomic sections in our benchmark programs are placed very carefully, they had little effect on performance. In the future, we plan to investigate microbenchmarks that demonstrate in what situations these optimizations are most useful.

9.2.3 Proposed Optimizations

Another potential optimization that we leave for future work involves the way that sequence numbers are allocated to threads. If it can be determined that two threads will never use the same shelter, then their sequence numbers need not be distinct. An analysis supporting such an optimization would likely rely on some form of a must-not-alias analysis [68].

9.3 Limitations

In the future, we plan to investigate the use of lightweight shape specifications that may allow our analysis to make use of a finer-grained shelter hierarchy to further improve usability and performance. That is, while our current system uses the finest-grained shelters possible given its simple analyses, it is not able to derive cases in which a programmer using explicit locks would exploit shape information, for example in the hand-over-hand locking of a tree or linked-list.

Furthermore, as the number of cores per socket increases, the scaling properties of our implementation will become more apparent. In moving from an 2-socket, 8-core machine to a 4-socket, 32-core machine we were able to expose and fix a number of performance bottlenecks. As the number of cores increases, and as their distribution across sockets changes, we expect that new bottlenecks may appear. On the other hand, as more cores share a single socket and cache, old bottlenecks may no longer be an issue.

One drawback of our current implementation is that it is highly tailored for an IA-32 architecture. That is, we use features specific to it, and assume cache coherency, and the guarantees made by its memory model. Our implementation would require some amount of adaptation in order to work on other architectures with different features and guarantees.

Finally, we have not yet thoroughly investigated the fairness properties of our system. Fairness and liveness issues did not arise in any of our benchmark programs, but in the future we wish to obtain more rigorous guarantees.

Part IV

Evaluation

Chapter 10

Evaluation of SharC

> The most essential characteristic of scientific technique is that it proceeds from experiment, not from tradition.
>
> *The Scientific Outlook*
> BERTRAND RUSSELL

In this chapter and the next we describe our experimental evaluation of SharC and shelter-based atomic sections. The purpose of both evaluations is two-fold. First, we wish to show that the runtime overhead of our analyses and mechanisms is not overly burdensome. Secondly, we wish to show that transitioning programs to use our extensions is a manageable task. The metric for evaluating runtime overhead is the amount of time it takes an application or benchmark to complete a task. The metric for evaluating the transition cost is the annotation burden placed on the programmer.[1]

10.1 SharC Evaluation

We applied SharC without Groups to 6 multi-threaded C programs totaling over 600k lines of code. Two of the programs were over 100k lines. These programs use threads in a variety of ways: some use threads for performance, whereas others use threads to hide I/O latency. Further, one benchmark is a server that spawns a thread for each client.

We found the following procedure for applying SharC to be expedient. Minimal changes are made to the source until the inference stage no longer results in ill-formed types. This involves removing casts that that incorrectly strip off our type qualifiers. Then, we run the program and inspect the error reports. These are usually sufficient to tell how data is shared in the program, and we use them to decide what objects are protected by which locks, and to note where the sharing mode of objects change (typically, SharC's sharing cast suggestions can be applied as is).

The goal of these experiments is to demonstrate that our approach is practical. In particular, it requires few enough code changes, and incurs low enough overhead that it could be used in

[1] A more salient piece of information might be how long it takes to add the needed annotations. We provide rough estimates; since our tools were developed as the benchmarks were annotated, it is difficult to provide precise data.

Benchmark					Time		Pagefaults		% `dynamic`
Name	Threads	Lines	Annots.	Changes	Orig.	SharC	Orig.	SharC	Accesses
pfscan	3	1.1k	8	11	1.84s	12%	21k	0.8%	80.0%
aget	3	1.1k	7	7	n/a	n/a	0.4k	30.8%	8.7%
pbzip2	5	10k	10	36	0.83s	11%	10k	1.6%	~0.0 %
dillo	4	49k	8	8	0.69s	14%	2.6k	78.8%	31.7 %
fftw	3	197k	7	39	44.1s	7%	63k	1.2%	0.2 %
stunnel	3	361k	20	22	0.39s	2%	0.5k	43.5%	~0.0%

Table 10.1: Benchmarks for SharC. For each test we show the maximum number of threads running concurrently (Threads), the size of the benchmark including comments (Lines), the number of annotations we added (Annots.) and other changes required (Changes). We also report the time and memory overhead caused by SharC along with the proportion of memory accesses to `dynamic` objects.

production systems. We found no serious errors[2] in our benchmarks because our tests only sought to exercise typical runs of the programs — we did not perform thorough regression testing.

Table 10.1 summarizes our experience using SharC. The reported runtimes are averages of 50 runs of each of the benchmarks. Memory overhead was measured by recording the number of minor pagefaults[3] incurred by each benchmark. All tests were run on a machine with a dual core 2GHz Intel® Xeon® 5130 processor with 2GB of memory. A total of 60 annotations, and 122 other code changes were required for the 600k lines of code. On average, SharC incurred a performance overhead of 9.2%, and a memory overhead of 26.1%.

The pfscan benchmark is a tool that spawns multiple threads for searching through files. It combines some features of grep and find. One thread finds all the paths that must be searched, and an arbitrary number of threads take paths off of a shared queue protected with a mutex and search files at those paths. Our test searched for a string in all files in an author's home directory. We found that running the test several times allowed all files to be held in the operating system's buffer cache, and so we were able to eliminate file systems effects in measuring the overhead.

The aget benchmark is a download accelerator. It spawns several threads that each download pieces of a file. We measured performance by downloading a Linux kernel tarball. The program was network bound, and so the overhead created by SharC was not measurable.

The pbzip2 benchmark is a parallel implementation of the block-based bzip2 compression algorithm. The benchmark consisted of using three worker threads to compress a 4MB file. The pbzip2 benchmark has threads for file I/O, and an arbitrary number of threads for (de)compressing data blocks, which the file-reader thread arranges into a shared queue. The functions that perform the (de)compression assume they have ownership of the blocks, and so we annotate their arguments as `private`. One benign race was found in a flag used to signal that reading from the input file has finished. At worst, a thread might yield an extra time before exiting.

[2] By "serious error" we mean a sharing strategy violation that causes unintended results.

[3] The number of minor pagefaults indicates the number of times the operating system kernel maps a page of the process's virtual address space to a physical frame of memory. It is a rough measure of memory usage.

The dillo benchmark is a web browser that aims to be small and fast. We measured the overhead of SharC by starting the browser, requesting a sequence of 8 URLs, and then closing the browser. The dillo benchmark uses threads to hide the latency of DNS lookup. It keeps a shared queue of the outstanding requests. Four worker threads read requests from the queue and initiate calls to `gethostbyname`. Several functions called from the worker threads assume that they own request data, so the arguments to these functions were annotated `private`. The memory overhead for dillo is higher because integers are cast to pointer type, and SharC infers they need to be reference counted. These bogus pointers are never dereferenced, but we incur minor pagefaults when their reference counts are adjusted. We suspect that this issue could be addressed if the programmer annotates the pointers that only store bogus values.

The fftw benchmark performs 32 random FFT's as generated by the benchmarking tool distributed with The Fastest Fourier Transform in the West [41]. The fftw benchmark computes by dividing arrays among a fixed number of worker threads. Ownership of arrays is transferred to each thread, and then reclaimed when the threads are finished. The functions that compute over the partial arrays assume that they own that memory, so it was only necessary to annotate those arguments as `private`.

The stunnel benchmark is a tool that allows the encryption of arbitrary TCP connections. It creates a thread for each client that it serves. The main thread initializes data for each client thread before spawning them. There are also global flags and counters, which are protected by locks. Stunnel uses the OpenSSL library, so it is also necessary to process it with SharC. Even though OpenSSL is not concurrent itself, SharC is able to verify that there are no thread-safety issues with its use by stunnel in our tests. Our experiments with stunnel involved encrypting three simultaneous connections to a simple echo server with each client sending and receiving 500 messages.

10.2 Conversion Effort

Our experiments on the programs above did not require extremely deep understanding. Locks tended to be declared near the objects they protected, threading-related code tended to be in a file called “thread.c”, `private` annotations were made close–both textually and in the call graph–to thread creation calls, and so forth. As in Section 4.1.1's example, SharC's error reports were often helpful in guiding annotation insertion. Also, SharC infers a reasonable default for omitted annotations. Therefore, we had no insurmountable problems in adding the few needed annotations. Time required to read and annotate relevant code varied between 2 and 4 hours. Time for reading and annotating larger codes did not grow proportionally because threading-related code tended to be concentrated in one place. Since the annotation burden is low, we do not believe that automating the annotation process would have a substantial benefit.

10.3 SharC with Groups

We applied SharC with Groups to 5 interesting applications, the largest of which approaches 1 million lines of code. All but one of these benchmarks use multithreading to improve performance;

Benchmark						Performance	
Name	Conc.	Lines	Anns.	Casts	Max Group	Orig.	SharC
ebarnes	2	3k	60	16	16384	6.26s	38%
em3d-s	2	1k	42	2	100000	3.59s	42%
knot	3	5k	60	17	75	0.69s	19%
eog	4	38k	67	38	20	1.42s	6%
GIMP	4	936k	37	32	8	9.67s	13%

Table 10.2: Benchmarks for SharC with Groups. For each test we show the maximum number of threads running concurrently (Conc.), the size of the benchmark including comments (Lines), the number of annotations we added (Anns.), sharing casts required (Casts), and the maximum size of a group in a run of the benchmark. We also report the time overhead caused by SharC.

we have included in our benchmarks a webserver that uses a thread pool to serve clients in parallel and hide I/O latency.

None of the benchmarks required more than a few days to make sufficient annotations to suppress a small number of false error reports and achieve good performance. The amount of human program understanding required to make these annotations is not especially burdensome.

The goal of these experiments is to show that the addition of Groups to SharC allows practical checking of data sharing in a substantially wider range of programs than it would be able to without groups, while maintaining low overhead. In the course of running our benchmarks, we found no bugs, and the only sharing errors found were benign data races on flags used for condition variables. These races would at worst cause a thread to needlessly call a `condition_wait()` function an extra time. For these flags we used SharC's `racy` annotation to suppress the false error reports. It is not surprising that we did not find more serious bugs because our testing of the benchmark applications was intended only to measure the performance overhead seen in typical use-cases.

10.3.1 Benchmarks

The results of our experiments are reported in Table 10.2. The figures we report are averages over 50 runs of each benchmark. These experiments were performed on a machine with a dual core 2GHz Intel® Xeon® 5130 processor with 2GB of memory. Excepting the scientific benchmarks, the runtime overhead imposed by SharC with Groups is less than 20%. The higher overhead in the scientific benchmarks is due to reference count updates in the inner loops. We also observed a reasonable annotation burden, ranging from one annotation or cast per 25 lines in the small benchmarks to less than one per 10,000 lines on the largest.

Ebarnes was presented in Section 4.3.5. Em3d-s is another adapted Split-C scientific program that models the propagation of electromagnetic waves through objects in three dimensions. In this simulation the system is modeled as a bipartite graph. One partition models the electric field, and the other partition models the magnetic field. At the beginning of the simulation the graph is constructed privately to one thread. Then, a sharing cast is used to indicate that the graph is

shared with other threads with accesses synchronized by barriers. To allow the sharing mode of the entire graph to change, the nodes and edges form a group whose leader is the structure containing pointers to the lists of nodes and edges.

Knot [90] is a simple multi-threaded webserver. It maintains a thread pool for processing client requests. The threads share an in-memory cache of the files on disk. The cache is implemented as a hash table and is protected by a lock. Each call to the hash table API is made while the lock is held. Therefore, we cast the entire hash table to `private` before passing it to the hash table functions. The primary advantage of this approach is that it allows the hash table API to be used both in single-threaded and multi-threaded contexts. The benchmark for knot involves two clients simultaneously making 10k requests for small files that fit entirely in the in-memory cache. The webserver and clients both ran on the same host. The goal of this setup is to minimize the extent to which the benchmark is I/O bound. The overhead for knot in Table 10.2 is reported as the percent increase in CPU utilization. However, runtime for the benchmark did not increase significantly, nor did the throughput decrease significantly. Because the sharing mode of the entire in-memory cache changes from locked to private and back, the hash table data structure used for the cache forms a single group along with all cache entries. The hash table object containing pointers to the array used for the hash table and other metadata is the group leader.

Eog is the Eye-of-Gnome image manipulation program distributed with the Gnome desktop environment. It can open, display, and rotate many different image formats. Eog uses threads to hide the latency of the operations performed on images from the user interface. These operations include loading and saving images, transforming them, and creating image thumbnails. The GUI places “Job” data structures in a lock protected queue. When worker threads take jobs off of the queue, they must cast the entire Job data structure, which contains the image, and image metadata, to the `private` mode. Each Job structure and the objects to which it has pointers are placed in a group with the Job structure itself being the group leader. For this benchmark we measured the CPU time needed to open an image, rotate it, and then save it back to disk.

The GIMP is also an image manipulation program, but with many more features than Eye-of-Gnome. In particular, it is script-able, and capable of many more sophisticated image transformations. On loading an image, the GIMP divides it up into distinct sections, which are stored in data structures called “Tiles.” To perform a transformation, the GIMP creates a thread pool and an iterator over tiles that is protected by a lock. When a tile is in the iterator it is in the locked mode, but when a thread removes it from the iterator it casts it to the `private` mode. The Tile object contains pointers that would be cumbersome to cast individually, so each Tile object and the objects to which it has pointers are made a group with the Tile object itself being the group leader. For this benchmark, we wrote a script that opened an image, performed 20 separate “Blur” operations, and then saved the image back to disk.

10.3.2 Scaling

In order to see how SharC with Groups scales with the number of threads, we ran the two scientific benchmarks on a server machine with 4GB of memory and two quad-core Intel® Xeon® E5462 processors running at 2.8GHz. As shown in Figure 10.1, we observed that the overhead incurred by our system did not increase significantly as the number of threads ranged from two to

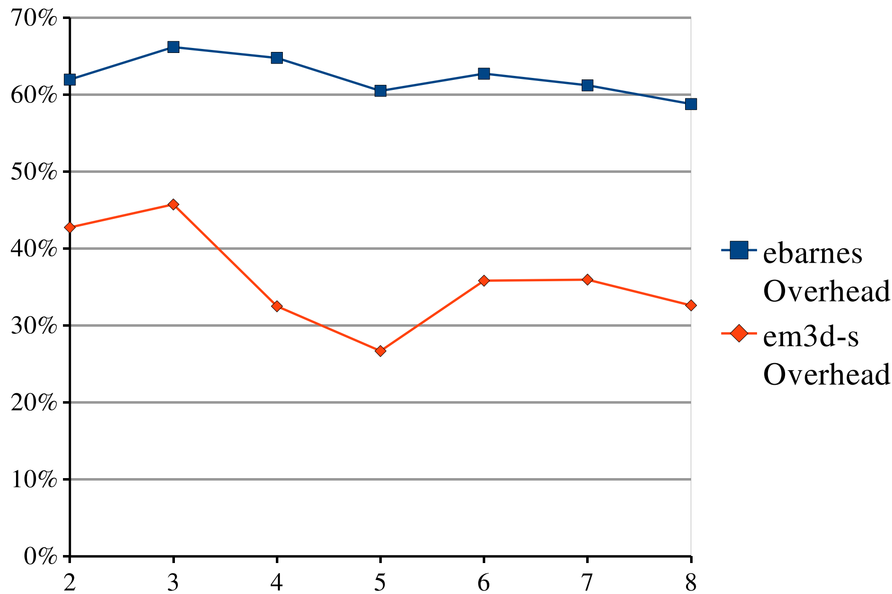


Figure 10.1: The overhead incurred by SharC with Groups as the number of threads increases and the workload remains fixed for the ebarnes and em3d-s benchmarks.

eight while the workload was held constant. The difference in relative overhead on this machine vs the dual core machine used in the main experiments is due to differential speedup in the application and overhead: absolute overhead for both ebarnes and em3d-s decreased by about 30%, as expected from the clock speed increase, while the application code sped up by 51%–59%, much more than the clock speed increase. We suspect, though have not confirmed, that the application code gets an extra advantage from the larger L2 cache (2x6MB vs 4MB) and memory bus speed (1600MHz vs 1333MHz).

The main scaling bottleneck in SharC is that only one thread may be computing a reference count at any time. This could impact programs performing many group or sharing casts. However, this effect was not observed in our experiments.

10.3.3 Sources of Overhead

Figure 10.2 shows the sources of performance overhead for our benchmarks. They are broken down into concurrent reference counting, the SharC runtime checks, and various infrastructure costs. Infrastructure costs are incurred by the use of the CIL [71] compiler front end[4], a custom malloc implementation needed for concurrent reference counting, and the need to null out pointers on allocation for soundness reasons.

We observe substantial reference counting overhead in the ebarnes and em3d-s benchmarks

[4]Using the CIL front end can cause changes in performance. CIL performs some transformations (e.g. introducing additional temporary variables) that can both enable and prevent certain C compiler optimizations.

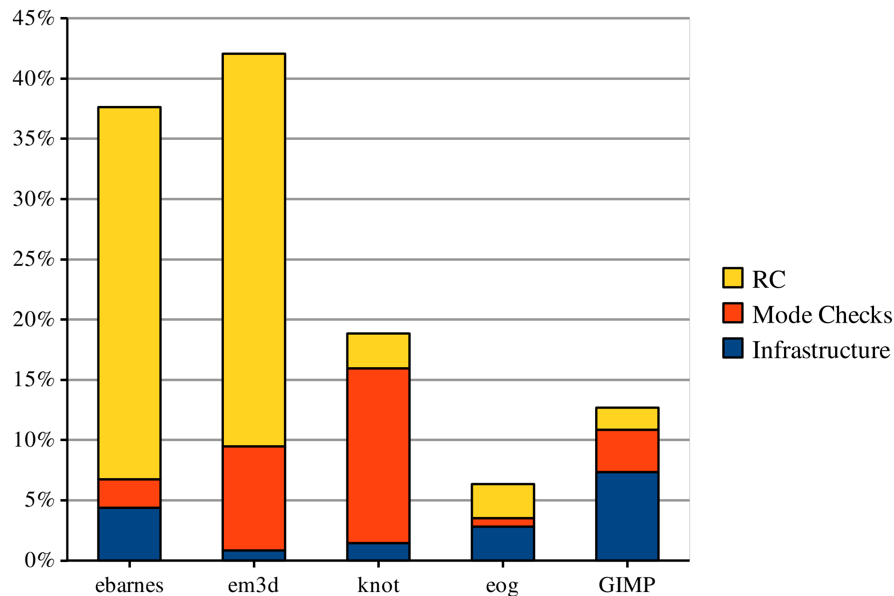


Figure 10.2: The breakdown of overheads in SharC. For each benchmark, we show the cost of reference counting(RC), the dynamic checks for sharing modes (Mode Checks), and the cost of using our compiler front-end and infrastructure.

due to the reference count updates needed when building and traversing the oct-trees and graph respectively. In eog and the GIMP, the overall low overhead makes the cost of our custom runtime more apparent.

10.3.4 The Need for Groups

In Table 10.2, we also report the maximum size of a group in each of our benchmarks. We interpret group size to be a rough measure of how costly it would be to port a program to SharC without groups, both in terms of performance and programmer effort. This interpretation is justified in the following ways. Without groups, it would be necessary to cast each object in a group individually. In order to meet the single reference requirement for the sharing cast, it is necessary to completely remove an object from a data structure before casting it. If a data structure has cycles, the code to perform this operation can be complex, and goes far beyond the low level of effort needed to annotate a program. Further, the additional data structure traversal, reconstruction, and reference counts would cause additional programmer and performance overhead.

Chapter 11

Evaluation of Shelters

All men by nature desire to know. An indication of this is the delight we take in our senses. The reason is that this makes us know and brings to light many differences between things.

Metaphysica
ARISTOTLE

We have modified a number of programs to use shelters as the mechanism for enforcing atomic sections, and we have measured the runtime performance of these programs on typical inputs. The purpose of this evaluation is to investigate the convenience of using shelter-based atomic sections, and to compare the runtime performance of our implementation with four other mechanisms for enforcing atomicity, namely explicit locking, software transactional memory, a single global lock, and shelters implemented with pthread reader-writer locks.

In the implementation of shelters using reader-writer locks, each shelter contains a lock. When registering for shelters at the beginning of atomic sections, the locks are sorted by address to avoid deadlock before being acquired. When a fine-grained shelter is registered, the shelter's lock is acquired in read mode if the section only reads the sheltered data, and in write mode otherwise. When a coarse-grained shelter is registered, the lock is acquired in write mode. A shelter's ancestors' locks are always acquired in read mode.

11.1 Experimental Setup

All of our experiments were performed on a 2.27GHz Intel® Xeon® X7560 machine with four processors each with eight cores having 32GB of main memory running Linux 2.6.18. We chose to compare against an Intel compiler for C/C++ that includes an STM implementation [80]. Other STM implementations may give better performance [15], but the Intel STM compiler has an annotation burden that is similar to that of our system, and requires no additional special hardware. We also used the Intel compiler with the STM features disabled as the back-end of our system, and to compile the other versions of the benchmarks. The compile-time analyses used for our system did not add significantly to compilation time.

11.1.1 Intel STM

The Intel STM compiler adds three relevant extensions to C: two statements and a function attribute.

- `__tm_abort` — This statement causes the enclosing transaction to abort.
- `__tm_atomic {} else {}` — Code inside of the first block runs as a transaction. If a `__tm_abort` statement executes in the first block, the transaction aborts and execution resumes in the `else` block if there is one.
- `tm_callable` — This function attribute must be placed on functions called inside of transactions so that the compiler knows to create a version of the function in which reads and writes are instrumented.

Two of the programs used condition variables and signaling. To accomplish this with the Intel STM, we used constructs like the following:

```
do {
  retry = 0;
  __tm_atomic {
   if (not(cond)) __tm_abort;
   /* proceed */
  } else { retry = 1; }
} while(retry);
```

Writes that change `cond` must also be made inside of a transaction, and the `else` branch of the `__tm_atomic` statement may optionally sleep instead of immediately retrying.

Occasionally, slight modifications were made to programs to avoid activation of higher levels of the shelter hierarchy where possible. We believe this practice is consistent with how a working programmer would improve the performance of a program using atomic sections, and the STM and explicit locking versions also benefited from these modifications because transactions were smaller, and locks were held for shorter periods of time, respectively. In the explicit locking versions, the locking was made as fine-grained as possible without substantially rewriting the programs.

11.2 Benchmarks

Our benchmark programs consist of the STAMP STM benchmark suite [15], along with: pbzip2, a parallel version of bzip2; pfscan, a parallel file scanning tool; and ebarnes, an n-body simulation using the Barnes-Hut algorithm [9]. Table 11.1 shows the size of each of the benchmark programs, the number of atomic sections in each, and the number of other annotations that were needed to use Intel's STM and our system, respectively. The other annotations for Intel's STM are the `tm_callable` annotations that must be placed on functions called from transactions. The other annotations for our system are the `sheltered_by` annotations and the `needs_shelters`

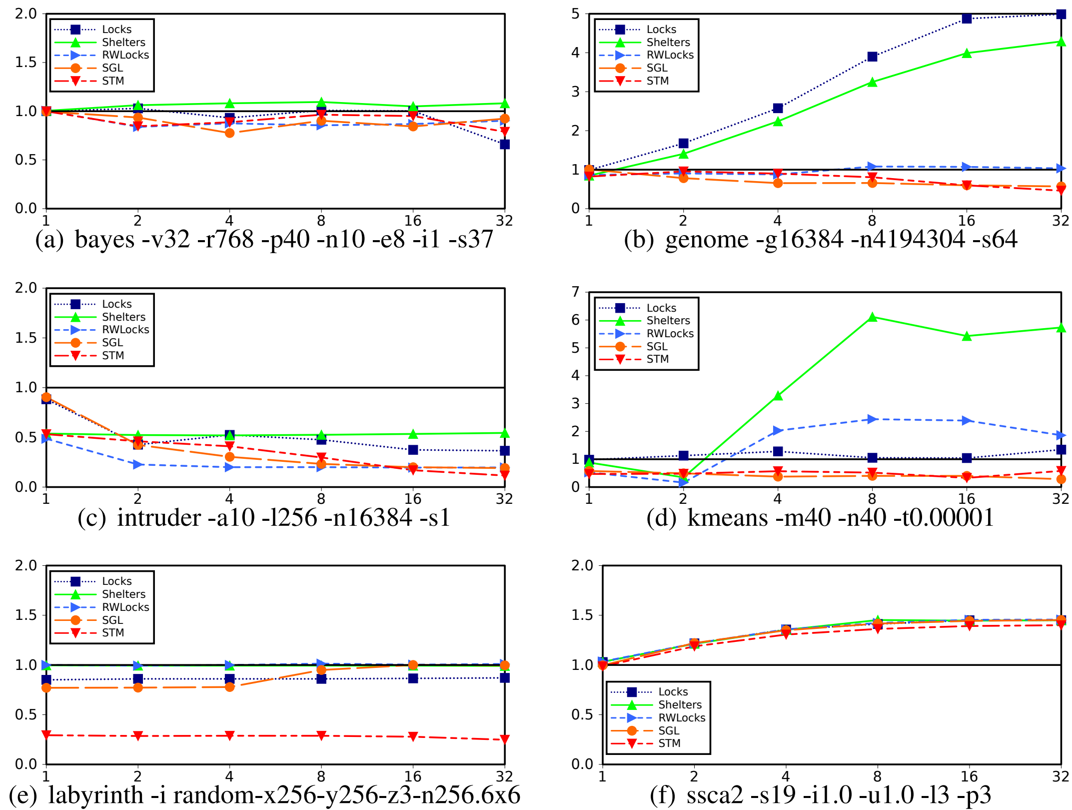
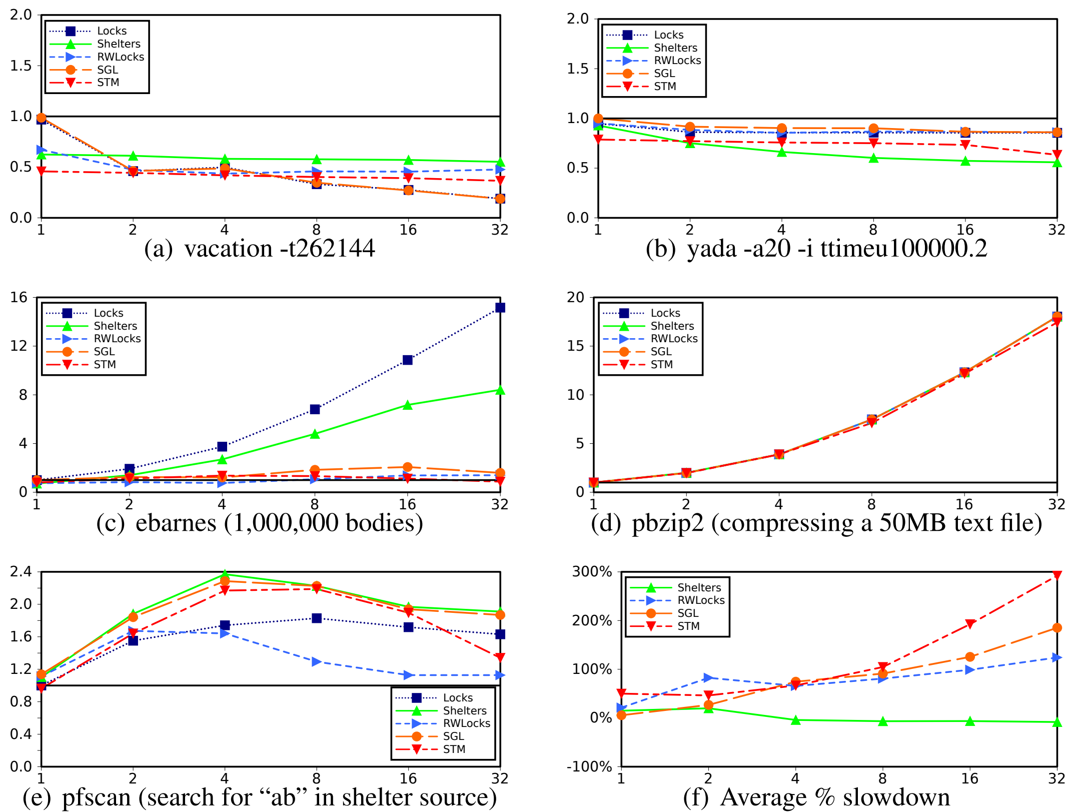


Figure 11.1: Graphs (a) - (f) show speedup over sequential runs versus the number of threads used for our benchmark programs when run with explicit locking (Locks), shelters (Shelters), Intel STM (STM), a single global lock (SGL), and shelters implemented with reader-writer locks (RWLocks). Higher is better.



(a) vacation -t262144

(b) yada -a20 -i ttimeu100000.2

(c) ebarnes (1,000,000 bodies)

(d) pbzip2 (compressing a 50MB text file)

(e) pfscan (search for “ab” in shelter source)

(f) Average % slowdown

Figure 11.2: Graphs (a) - (e) show speedup over sequential runs versus the number of threads used for our benchmark programs when run with explicit locking (Locks), shelters (Shelters), Intel STM (STM), a single global lock (SGL), and shelters implemented with reader-writer locks (RWLocks). Higher is better. Graph (f) shows average percent slowdown with respect to explicit locking over all benchmarks versus the number of threads. Lower is better.

Name	Size (kloc)	Atm. sects.	STM Annts.	Shelter Annts.	Seq. Time
bayes	12.0	15	47	42	9.97s
genome	10.0	5	16	25	8.58s
intruder	11.3	3	61	64	2.26s
kmeans	3.9	3	4	7	9.17s
labyrinth	8.2	3	50	46	3.02s
ssca2	9.2	1	5	12	9.73s
vacation	11.0	3	159	122	1.53s
yada	13.4	6	105	86	4.19s
ebarnes	13.4	3	8	9	16.07s
pbzip2	10.0	10	4	14	10.46s
pfscan	2.8	6	4	11	2.56s
total	105.2	48	463	438	

Table 11.1: Program size, number of atomic sections, number of annotations for Intel STM and Shelters, and sequential runtime for our benchmark programs.

function annotations. The STAMP benchmarks are distributed with the `tm_callable` annotations already placed, some of which are redundant. We also made redundant annotations when adapting the programs for our system, as we also found the annotations to be useful for documentations purposes. The annotation counts in the table include these redundant annotations. This is why in the STAMP benchmarks the annotation count for STM is sometimes higher than it is for our system.

The results of our experiments are given in the graphs of Figure 11.2. The graphs show speedup over sequential runs (i.e. $T_{sequential}/T_{parallel}$) versus the number of cores used. Each reported result is the average of at least 50 runs. Command line arguments passed to the STAMP benchmarks are also given in the captions.

STAMP Benchmarks

The intended usage of the STAMP benchmark suite is to compare different transactional memory implementations. In addition, we believe that it is also a suitable benchmark suite for the more general task of comparing different implementations of atomic sections.

The bayes benchmark implements an algorithm used in learning Bayesian networks. It involves multiple threads concurrently modifying and searching in a graph. The bayes benchmark required activation of shelters higher in the hierarchy for the linked lists used to represent the graph adjacency list. The performance of the bayes benchmark is shown in Figure 11.1(a). None of the implementations were able to yield any significant parallel speedup. For our system and explicit locking, lock and shelter contention is high because exclusive access to the entire graph is acquired for each atomic section. For the STM runs, data contention and big transactions caused many conflicts and rollbacks, limiting scaling.

The genome benchmark reconstructs a gene sequence from overlapping fragments. It involves multiple threads concurrently adding elements to hashtables, and searching in and modifying strings. The performance of the genome benchmark is shown in Figure 11.1(b). The explicit locking and shelters versions scale up as threads are added, but the STM version does not due to the overhead of memory barriers for memory reads and writes in atomic sections [17].

The intruder benchmark implements a signature-based intrusion detection algorithm. Packet streams are concurrently collected in a red-black tree, added to a queue when finished, and examined for a matching signature. The performance of the intruder benchmark is shown in Figure 11.1(c). None of the implementations achieve a parallel speedup due to being serialized by access to the red-black tree.

The kmeans benchmark implements a clustering algorithm. Threads concurrently read and modify several arrays used to calculate the means of, and membership in, the clusters.The performance of the kmeans benchmark is shown in Figure 11.1(d). Shelters and shelters implemented with reader-writer locks are able to achieve some parallel speedup. The STM implementation is penalized by the necessity of instrumenting memory reads and writes in transactions as for the genome benchmark. Explicit locks are penalized by the inability to provide concurrent read access, however this could be fixed by using reader-writer locks.

The labyrinth benchmark implements an algorithm for navigating a maze. Threads concurrently update a two-dimensional array to add to it paths through the maze. The performance of the labyrinth benchmark is shown in Figure 11.1(e). None of the implementations are able to achieve a parallel speedup due to being serialized by access to the maze array. The STM version is again penalized by instrumentation of reads and writes.

The ssca2 benchmark implements a graph kernel used in a number of different algorithms. Threads concurrently update graph adjacency lists. The performance of this benchmark is shown in Figure 11.1(f). Each implementation is able to achieve a modest parallel speedup.

The vacation benchmark implements a travel reservation system. Threads concurrently access and modify relations stored as maps represented by red-black trees. The vacation benchmark required activation of the higher levels in the shelter hierarchy for linked lists stored in the maps. The performance of this benchmark is shown in Figure 11.2(a). None of the implementations are able to achieve any parallel speedup due to being serialized by access to the red-black trees.

The yada benchmark implements a mesh refinement algorithm. Threads concurrently add and remove nodes and edges from the mesh. As with the bayes benchmark, the yada benchmark required activation of higher levels in the shelter hierarchy for graph adjacency lists. The performance of this benchmark is shown in Figure 11.2(b). None of the implementations are able to achieve a parallel speedup. Even though the mesh is large and triangles in need of refinement begin distributed uniformly throughout the mesh, as the algorithm runs, the location of triangles in need of refinement becomes correlated. This increases both data and lock contention, which prevents scaling [59]. The performance of our system implementation degrades a bit less gracefully than the other implementations.

Application Benchmarks

In addition to the STAMP benchmarks, we chose two multi-threaded C applications using explicit locking to protect shared objects. We chose these applications in order to compare the

real-world performance of shelters with the other implementations. We also included a benchmark implementing an n-body simulation using the Barnes-Hut algorithm. We chose it because the parallel speedup in the oct-tree-building phase is very sensitive to the synchronization strategy.

The ebarnes benchmark is an n-body simulation adapted from the Barnes-Hut Splash2 benchmark [92]. In each phase of the simulation it builds an oct-tree in parallel and uses it to update the positions of the bodies. For this benchmark, 1 million bodies were simulated so that building the oct-tree in parallel would give a significant performance advantage. The performance of this benchmark is shown in Figure 11.2(c). Shelters and explicit locks allow scaling up as cores are added, whereas the other implementations fail to allow scaling.

The pbzip2 benchmark is a parallel implementation of the popular block-based compression algorithm. Threads take blocks to compress off of a shared queue. Separate threads add blocks to the queue, and write the compressed blocks to a file. For this benchmark, a 50MB text file was compressed. The file was small enough to fit into the operating system's file system caches. The performance of this benchmark is shown in Figure 11.2(d). Each implementation manages to obtain a parallel speedup.

The pfscan benchmark is a tool that searches for a string in all files under a given directory tree. One thread places paths to files to search on a queue while other threads takes paths off the queue and search for the string in the files. For this benchmark, we searched for the string "ab" in the Linux source code tree. The tree was small enough to fit into the operating system's file system caches. The performance of this benchmark is shown in Figure 11.2(e). Each implementation manages to obtain a parallel speedup until calls into the operating system and limited workload size impede further performance gains.

11.2.1 Effects of Workload Size

We also did an experiment in which the number of bodies simulated in the ebarnes benchmark was varied from 100k bodies, which fit comfortably in the caches, to 8 million bodies, which exceeded the capacity of the caches. We ran the simulations with each implementation, and on 4, 8, 16, and 32 cores. We did not observe any changes in relative overhead with respect to the explicit locking version as workload size increased.

11.2.2 Discussion

Our system's implementation manages to obtain performance comparable to explicit locking while having most of the convenience of a mature STM implementation. The graph in Figure 11.2(f) shows the average percent slowdown over all of the benchmarks of the shelters, single global lock, reader/writer locks, and STM runs with respect to the locking runs versus the number of threads used. Our system's implementation scales up where possible as threads are added in cases where the the other implementations fail to do so.

Chapter 12

Conclusion and Future Work

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> *Computing Machinery and Intelligence*
> ALAN TURING

This dissertation presented SharC and Shelters, extensions to the C language enabling safe and efficient concurrent systems-level programming. These extensions were integrated into the Ivy compiler not only so that they could benefit from the guarantees of Ivy's other components, but also so that they could render these other components sound when presented with multi-threaded programs. Furthermore, SharC and Shelters benefited greatly from Ivy's design philosophy. In particular, Ivy's emphasis on modularity and gradual evolution allowed us to apply our new extensions to large real-world programs. Applying SharC and Shelters to these programs was a straightforward process, and resulted in code having a small runtime cost in exchange for Ivy's improved safety guarantees.

12.1 Future Work

Ivy strikes a unique balance among verification power, programmer interaction, and efficiency. The specific techniques used by Deputy, Heapsafe, SharC, and Shelters have the potential to be useful for languages beyond C, however Ivy's design philosophy of simple and sparse annotations, interoperability, and modularity are likely more widely applicable. Given that, here are a few potential directions for future work.

- Ivy-like annotations could describe the correct use of collections APIs. For example, when removing a buffer from a queue, it could be specified by way of simple annotations whether the removing thread receives the only reference to the object, whether it must deallocate memory for the removed object, or whether a lock is held for the removed object that must be released, and so forth. A mixed static/dynamic analysis could then verify that these APIs, and the objects managed by the collections are used correctly.

- Further, SharC's techniques could be useful for efficiently debugging CUDA [24] programs. This could have significant impact as CUDA is widely used, but lacks tool support due to its relative newness.

- The nesC language for embedded systems programming already makes use of Deputy. Extending it with features from SharC and Shelters could enable it to also work well for more general multi-threaded, event-based, systems programming, such as in a traditional operating system.

- Java, C++, and Objective-C are similar to C, and many techniques from Ivy would apply to them well. Figuring out how to deal well with the more pervasive polymorphism in these languages with respect to Ivy's annotations would be an interesting contribution.

- Many of Ivy's static analyses could be made more precise. However, doing this while still avoiding whole-program analysis could provide an interesting challenge.

Bibliography

[1] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *ASE'05*.

[2] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[3] E. Allen, D. Chase, V. Luchangco, J.-W. Maessenand S. Ryuand G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress language specification version 1.0*, 2008. http://research.sun.com/projects/plrg/fortress.pdf.

[4] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA'05*, pages 316–327.

[5] Zachary Anderson and David Gay. Atomic shelters: Coping with multi-core fallout. Technical Report UCB/EECS-2010-39, EECS Department, University of California, Berkeley, Apr 2010.

[6] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *PLDI'08*, pages 149–158.

[7] Zachary Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *PLDI'09*, pages 98–109, New York, NY, USA, 2009. ACM.

[8] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI'94*.

[9] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[10] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HotPar'09*.

[11] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT.

[12] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02*, pages 211–230.

[13] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *OOPSLA'03*, pages 213–223.

[14] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *PLDI'03*, pages 324–337.

[15] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08*.

[16] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

[17] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[18] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[19] 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 *SPAA'98*, pages 298–309.

[20] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI'08*.

[21] 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 *PLDI'02*, pages 258–269.

[22] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *ESOP'07*.

[23] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL'99*, pages 262–275.

[24] NVIDIA CUDA. *Compute Unified Device Architecture*. http://www.nvidia.com/object/cuda_develop.html.

[25] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI'01*, pages 59–69, New York, NY, USA, 2001. ACM.

[26] developers.sun.com. LockLint - static data race and deadlock detection tool for C. http://developers.sun.com/solaris/articles/locklint.html.

[27] Ulrich Drepper. Futexes are tricky, 2009. http://people.redhat.com/drepper/futex.pdf.

[28] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *OOPSLA'04*.

[29] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI'07*, pages 245–255.

[30] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*, pages 237–252.

[31] Cormac Flanagan and Martin Abadi. Object types against races. In *Conference on Concurrent Theory (CONCUR)*, 1999.

[32] Cormac Flanagan and Martin Abadi. Types for safe locking. In *ESOP'99*, 1999.

[33] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*, pages 256–267.

[34] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI'00*, pages 219–232.

[35] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI'05*, pages 47–58.

[36] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI'03*, pages 338–349.

[37] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI'99*, pages 192–203.

[38] Free Software Foundataion. GCC, the gnu compiler collection. http://gcc.gnu.org.

[39] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[40] freedesktop.org. Gstreamer: Open source multimedia framework. http://gstreamer.freedesktop.org/.

[41] Matteo Frigo. A fast Fourier transform compiler. In *PLDI'99*, pages 169–180.

[42] David Gay and Alex Aiken. Language support for regions. In *PLDI'01*, pages 70–80.

[43] David Gay, Rob Ennals, and Eric Brewer. Safe manual memory management. In *ISMM'07*, pages 2–14.

[44] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI'03*, pages 1–11, New York, NY, USA, 2003. ACM.

[45] Jonathan Goodman, Albert G. Greenberg, Neal Madras, and Peter March. Stability of binary exponential backoff. *J. ACM*, 35(3):579–602, 1988.

[46] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the assured evolution of concurrent Java programs. *Sci. Comput. Program.*, 58(3):384–411, 2005.

[47] Dan Grossman. Type-safe multithreading in Cyclone. In *TLDI'03*.

[48] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[49] John L. Henning. Performance counters and development of spec cpu2006. *SIGARCH Comput. Archit. News*, 35(1):118–121, 2007.

[50] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI'04*, pages 1–13.

[51] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Inferring locking for atomic sections. In *TRANSACT'06*.

[52] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ASPLOS'00*.

[53] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *MSPC'06*.

[54] Intel. *Intel C++ STM Compiler Prototype Edition 3.0*, 2008.

[55] Intel Corp. *Intel 64 Architecture Memory Ordering White Paper*, 1.0 edition, August 2007.

[56] kernel.org. Kernel bug tracker. http://bugzilla.kernel.org/.

[57] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[58] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *SUPERCOM'93*, pages 262–273.

[59] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI'07*, pages 211–222.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[61] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.

[62] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1):1–69, 2006.

[63] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[64] V.J. Marathe, M.F. Spear, C. Heriot, A.Acharya, D. Eisenstat, W.N. Scherer III, and M.L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT'06*.

[65] Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. Dynamically checking ownership policies in concurrent C/C++ programs. In *POPL'10*. Full version, preprint.

[66] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pages 346–358.

[67] mozilla.org. Bugzilla@mozilla bug tracker. http://bugzilla.mozilla.org/.

[68] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *PLDI'07*, pages 327–338.

[69] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI'06*, pages 308–319.

[70] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[71] George C. Necula, Scott McPeak, and Westley Weimer. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pages 213–228. http://cil.sourceforge.net/.

[72] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP'03*, pages 167–178.

[73] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. volume 38, pages 179–190, 2003.

[74] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI'06*, pages 320–331.

[75] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In *PLDI'04*, pages 14–24.

[76] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA'05*, pages 494–505.

[77] Derek Rayside, Lucy Mendel, and Daniel Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA'06*, pages 57–64.

[78] Martin C. Rinard and Monica S. Lam. Semantic foundations of Jade. In *POPL '92*, pages 105–118.

[79] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[80] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06*, pages 187–197.

[81] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP'05*, pages 83–94.

[82] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP'97*, pages 27–37.

[83] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, pages 166–182.

[84] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC'95*.

[85] Nir Shavit and Dan Touitou. Software transactional memory. 1995.

[86] SPEC. Standard Performance Evaluation Corporation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95*, July 1995.

[87] Tachio Terauchi. Checking race freedom via linear programming. In *PLDI'08*, pages 1–10.

[88] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. In *Information and Computation*, volume 132, pages 109–176, 1977.

[89] US-CERT. Technical cyber security alerts. http://www.us-cert.gov/cas/techalerts/index.html.

[90] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP'03*, pages 268–281.

[91] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *ESEC-FSE'07*, pages 205–214.

[92] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA'95*, pages 24–36.

[93] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP'05*, pages 221–234.

Part V

Appendix

Appendix A

The Soundness of SharC with Groups

Our basic approach is to prove, by induction over the operational semantic steps that the following two properties hold at all times:

- The statements in all threads are well-typed. In particular, our typing rules require that runtime checks necessary before performing an assignment either hold or are yet to be performed (Section A.2).
- The memory and thread environments are consistent (Section A.3).

From these properties it is easy to prove that private cells are only accessed by their owner, that locked cells are only accessed when the lock is held, and that there are no data-races on dynamic cells.

The proof consists of a few general-usage lemmas (Section A.3.1), proofs that single-thread steps, thread creation and thread destruction preserve the two properties (Sections A.3.2 through A.5), and a final section that puts all the pieces together to prove that the statically declared sharing modes are respected (Section A.6).

A.1 Preliminaries

We use *address* to refer to an element of the domain of a memory, and note that 0 is never a valid address ($0 \notin \mathrm{dom}(M)$). We use *cell* to refer to an actual memory element $M(a)$. An *lvalue* is an expression that denotes a field of a particular cell.

We use letters $a - d$ to refer to addresses, $i - n$ to refer to integers and all other letters to refer to identifiers. Normally, letters $f - h$ refer to fields, $x - z$ to variables, and $t - v$ to structure type names.

A.2 Runtime Typing

To show that our operational semantics preserve types, we show that programs remain well-typed at all points during execution. To do this, we need typing rules that enforce the presence of

$\Gamma \vdash s \Rightarrow s'$	In environment Γ statement s compiles to s', which is identical to s except for added runtime checks.

$$\text{(SEQ)}\ \frac{\Gamma \vdash s_1 \Rightarrow s_1' \qquad \Gamma \vdash s_2 \Rightarrow s_2'}{\Gamma \vdash s_1; s_2 \Rightarrow s_1'; s_2'} \qquad \text{(SPAWN)}\ \frac{f \in F}{\Gamma \vdash \text{spawn } f() \Rightarrow \text{spawn } f()}$$

$$\text{(LOCK)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash \text{lock } x \Rightarrow \text{lock } x} \qquad \text{(UNLOCK)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash \text{unlock } x \Rightarrow \text{unlock } x}$$

Figure A.1: Elided rules. F is the set of all thread functions in the program.

$M, id \models \ell : \tau, \omega$	In memory M and thread id, ℓ is a well-typed expression with type τ assuming ω holds.

$$\text{(NAME-R)}\ \frac{M_\rho(id.x) = m<y>\ t}{M, id \models x : m<y>\ t, \epsilon} \qquad \text{(SAME FIELD-R)}\ \frac{M_\rho(id.x) = m<y>\ t \quad T(t.f) = t'}{M, id \models x.f : m<y>\ t', m(x)} \qquad \text{(OTHER FIELD-R)}\ \frac{M_\rho(id.x) = m<y>\ t \quad T(t.f) = m'<g>\ t'}{M, id \models x.f : m'<x.g>\ t', m(x)}$$

Figure A.2: Runtime typing judgments for expressions.

runtime checks rather than add them, and handle the runtime-only statements (**skip**, **wait**). These typing rules are given in Figure A.3 and are derived from the typing judgments of Figure 5.4, completed by the rules in Figure A.1.

$M, id, r \models s$ checks that s is a well-typed statement of thread id in memory M. Runtime checks are special: if r is false, the checks for an assignment must all be present in its when clause. If r is true, then a prefix of the necessary checks can instead hold in M and be omitted from the when clause. For soundness, we must only check an assignment with r true when it is the first statement of a thread: this is enforced by passing false for r when checking s_2 in SEQ-R. For convenience, we write $M, id \models s$ to stand for $M, id, \text{true} \models s$.

A.3 Consistency

To ensure that programs remain type-safe, we need to know that types and owners in the memory are consistent. Furthermore, to avoid group and single-object casts changing the types of local variables, we must assert that the memory cell containing a thread's environment is unaddressable.

Definition 5 *Memory consistency. M is consistent* with threads $id_1, \ldots, id_n$, written $id_1, \ldots, id_n \models$

$M, id, r \models s$	In memory M and thread id, statement s is well-typed. If r is true, valid runtime conditions may be assumed.

(LOCK-R)
$$\frac{M_\rho(id.x) = \tau}{M, id, r \models \text{lock } x}$$

(UNLOCK-R)
$$\frac{M_\rho(id.x) = \tau}{M, id, r \models \text{unlock } x}$$

(DONE-R)
$$\frac{}{M, id, r \models \textbf{done}}$$

(SKIP-R)
$$\frac{}{M, id, r \models \textbf{skip}}$$

(SPAWN-R)
$$\frac{}{M, id, r \models \text{spawn } f()}$$

(SEQ-R)
$$\frac{M, id, r \models s_1 \qquad M, id, \text{false} \models s_2}{M, id, r \models s_1; s_2}$$

(NEW-R)
$$\frac{M_\rho(id.x) = m\text{<}y\text{>}\ t \qquad \mathrm{D}(M, id, x) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i}{M, id, r \models x := \text{new when } \omega_k, \ldots, \omega_n}$$

(GCAST-R)
$$\frac{x \neq z \qquad M_\rho(id.x) = m\text{<}y\text{>}\ t \qquad M_\rho(id.z) = m'\text{<}z\text{>}\ t \qquad \mathit{oneref}(z), \mathrm{D}(M, id, x), \mathrm{D}(M, id, z) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i}{M, id, r \models x := \text{gcast } z \text{ when } \omega_k, \ldots, \omega_n}$$

(SCAST-R)
$$\frac{x \neq z \qquad M_\rho(id.x) = m\text{<}w\text{>}t \qquad M_\rho(id.z) = m'\text{<}y\text{>}t' \qquad \mathit{oneref}(z), \mathrm{NF}(M, id, z), \mathrm{D}(M, id, x), \mathrm{D}(M, id, z) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i}{M, id, r \models x := \text{scast } z \text{ when } \omega_k, \ldots, \omega_n}$$

(READ-R)
$$\frac{M_\rho(id.x) = m\text{<}x'\text{>}\ t \qquad M, id \models \ell : m\text{<}\ell'\text{>}\ t, \omega \qquad \mathrm{R}(\omega), x'[\ell/x] = \ell', \mathrm{D}(M, id, x) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i}{M, id, r \models x := \ell \text{ when } \omega_k, \ldots, \omega_n}$$

(WRITE-R)
$$\frac{M, id \models x.f : m\text{<}\ell\text{>}\ t, \omega \qquad M_\rho(id.y) = m\text{<}y'\text{>}\ t \qquad \mathrm{W}(\omega), \ell[y/x.f] = y', \mathrm{D}(M, id, x.f) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i}{M, id, r \models x.f := y \text{ when } \omega_k, \ldots, \omega_n}$$

Figure A.3: Runtime typing judgments.

$$D(M, id, x) = \{y = \text{null} \mid x \neq y \wedge M_\rho(id.y) = m{<}x{>}\ t\}$$
$$D(M, id, x.f) = \{x.g = \text{null} \mid M_\rho(id.x) = m{<}y{>}\ t \wedge f \neq g \wedge t(g) = m{<}f{>}\ t\}$$
$$NF(M, id, x) = \{x.f = \text{null} \mid M_\rho(id.x) = m{<}y{>}t \wedge t(f) = t'\}$$
$$R(m(x)) = \begin{cases} chkread(x) \textbf{ if } m = \texttt{dynamic}; \\ m(x) \textbf{ otherwise} \end{cases} \quad W(m(x)) = \begin{cases} chkwrite(x) \textbf{ if } m = \texttt{dynamic}; \\ m(x) \textbf{ otherwise} \end{cases}$$

Figure A.4: Runtime checks for assignments.

M, if all thread identifiers are distinct, cell id_i is unaddressed and owned by id_i, and types and owners are consistent when an lvalue refers to a cell. Formally for all threads id_i

1. $id_i \neq 0$, and $i \neq j \Rightarrow id_i \neq id_j$ (threads are distinct)
2. $M_\rho(id_i) = \texttt{private}{<}0{>}\ t$ (thread environments not in groups)
3. $M_o(id_i) = id_i$ (thread environments are thread-owned)
4. $\nexists a.f(M_v(a.f) = id_i)$ (thread environments are unaddressed)

and for all addresses a where $M_\rho(a) = m{<}b{>}\ t$ with $t = (f_1 : \phi_1, \ldots, f_n : \phi_n)$:

5. if $c = M_v(a.f_i) \neq 0$ then
   (a) if $\phi_i = v$ then $M_\rho(c) = m{<}b{>}\ v$
   (b) if $(\phi_i = \texttt{private}{<}f_j{>}\ v) \vee (\phi_i = v \wedge m = \texttt{private})$ then $M_o(c) = M_o(a)$
   (c) if $\phi_i = m'{<}f_j{>}\ u$ then $M_\rho(c) = m'{<}M_v(a.f_j){>}\ u$
6. if $\phi_i = \texttt{private}{<}f_j{>}\ u$ then $b = 0 \wedge m = \texttt{private}$, and

A.3.1 Basic Properties

Lemma 1 *Thread sets and consistency.* If $id_1, \ldots, id_n \models M$ then $id \models M$ If $id_1, \ldots, id_n \models M$, $id' \models M$ and $\forall i.id' \neq id_i$ then $id_1, \ldots, id_n, id' \models M$

Lemma 2 *Private accesses by owning thread only.* Assume

$$id \models M \qquad M_\rho(a) = \texttt{private}{<}b{>}\ t$$

If $M, id : s \xrightarrow{s} M'$ causes thread id to access (read or write) cell a then $M_o(a) = id$
Corollary: If $M_o(a) \neq id$ then $M'_v(a) = M_v(a)$
Proof: By inspection of the operational semantic rules. First, all accesses to $M_v(id)$ are safe as $id \models M$ implies $M_o(id) = id$. Second, all accesses to cells $a = M_v(id.x)$ (due to lvalue $x.f$) are safe as $M_\rho(a) = \texttt{private}{<}b{>}\ t$ implies $M_\rho(id.x) = \texttt{private}{<}y{>}\ t$, which itself implies that $M_o(a) = M_o(id) = id$ (both from $id \models M$).

Lemma 3 *Locked accesses by locking thread only.* Assume

$$id \models M \quad M, id \models s \quad M_\rho(a) = \texttt{locked}<b>\ t$$

If $M, id : s \xrightarrow{s} M'$ causes thread id to access (read or write) cell a then $M_L(a) = id$

Corollary: If $M_L(a) \neq id$ then $M'_v(a) = M_v(a)$

Proof: By inspection of the operational semantic rules. First, all accesses to $M_v(id)$ are safe as $id \models M$ implies $M_\rho(id) = \texttt{private}<c>\ u$. Second, if thread id accesses cell $a = M_v(id.x)$ due to lvalue $x.f$, then $M_\rho(a) = \texttt{locked}<b>\ t$ implies $M_\rho(id.x) = \texttt{locked}<y>\ t$ (from $id \models M$). Inspection of the READ-R and WRITE-R typing rules (which check the only assignments that can access a) shows that $\omega_1 = \texttt{locked}(x)$, i.e. $\texttt{locked}(x)$ is checked first. Thus when the assignments' subsequent runtime checks, or the assignmemt itself access a, $M, id \models \texttt{locked}(x)$ and hence $M_L(a) = id$.

Lemma 4 *Dynamic access exclusive to writing thread only.* Assume

$$id \models M \quad M, id \models s \quad M_\rho(a) = \texttt{dynamic}<b>\ t$$

If $M, id : s \xrightarrow{s} M'$ causes thread id to write cell a, then $M_R(a) \subseteq M_W(a) = \{id\}$.

Corollary: If $M, id : s \xrightarrow{s} M'$ causes thread id to read cell a, then $id \in M_R(a)$.

Proof: Again, by inspection of the operational semantic rules, proceeding much as the proof of Lemma 3 except that in the WRITE-R typing rule, we are guaranteed that $chkwrite(a)$ is checked before the write, so $M, id \models chkwrite(a)$, and thus $M_R(a) \subseteq M_W(a) = \{id\}$. If s had caused a read, then the READ-R would ensure a $chkread(a)$ check before the read, and thus that $id \in M_R(a)$.

Lemma 5 *Lvalue types are respected.* If

$$\begin{gathered} id \models M \quad M, id \models \ell : m<\ell'>\ t, \omega \\ lval(M, id, \ell) = a.f \quad c = M_v(a.f) \neq 0 \quad lval(M, id, \ell') = b.g \end{gathered}$$

then

$$M_\rho(c) = m<M_v(b.g)>\ t \quad m = \texttt{private} \Rightarrow M_o(c) = id$$

Proof: Follows from the definition of memory consistency, and clause 6 of $id \models M$.

Lemma 6 *Dependent types safe under assignment.* If

$$\begin{gathered} id \models M \quad \mathrm{D}(M, id, \ell) = \omega_1, \ldots, \omega_n \quad \forall i. M, id \models \omega_i \\ lval(M, id, \ell) = a.f \quad M' = M[a.f \xrightarrow{v} b] \end{gathered}$$

then $\forall g \neq f. M'_\rho(a.g) = m<h>\ t \wedge M'_v(a.g) \neq 0 \Rightarrow M'_\rho(M'_v(a.g)) = m<M'_v(a.h)>\ t$ (clause 5c preserved for all fields but f)

Proof: Trivial for all cases except $h = f$. When $h = f$, $a.g = \text{null} \in \mathrm{D}(M, a, f)$ and hence $M_v(a.g) = 0 = M'_v(a.g)$, so the lemma holds.

Lemma 7 *Thread steps preserve local variable types* If

$$id \models M \quad M, id : s \xrightarrow{s} M'$$

then $M'_\rho(id) = M_\rho(id)$
Proof: Only *gcast* changes existing cell types, and only for cells whose group is non-zero. As $M_\rho(id) = \texttt{private<0>}\ t$, $M'_\rho(id) = M_\rho(id)$.

Lemma 8 *Other-thread steps preserve local variable values.* If

$$id, id' \models M \quad M, id' : s \xrightarrow{s} M'$$

then $M'_v(id.x) = M_v(id.x)$
Proof: $M_\rho(id) = \texttt{private<0>}\ t$ and $M_o(id) = id$, so by Lemma 2 $M'_v(id.x) = M_v(id.x)$.

Lemma 9 *Prototype thread preservation.* If

$$id_f \models M \quad \models M[id_f \xrightarrow{o} 0]$$
$$id_f \models M' \quad M'_\rho(id_f) = M_\rho(id_f) \quad M'_v(id_f) = M_v(id_f)$$

then $\models M'[id_f \xrightarrow{o} 0]$
Proof: Let $M'' = M'[id_f \xrightarrow{o} 0]$. Consider any lvalue $a.g$ in M'' with $c = M''_v(a.g) = M'_v(a.g)$ and $\phi = M''_\rho(a.g) = M'_\rho(a.g)$. If $c = 0$ then clause 5 holds. Otherwise, $id_f \models M'$ implies $c \neq id_f$, so $M''_\rho(c) = M'_\rho(c)$ and $M''_o(c) = mo'(c)$. For $a \neq id_f$, $M''_o(a) = M'_o(a)$, so $id_f \models M'$ implies that clause 5 holds for lvalue $a.g$. If $a = id_f$ $c = M_v(id_f.g)$ and $\phi = M_\rho(id_f.g)$. If $\phi = v \vee \phi = \texttt{private<}g\texttt{>}\ t$ then $M_o(c) = M_o(id_f) = id_f$ and $M_o(c) = M[id_f \xrightarrow{o} 0]_o(id_f) = 0$, a contradiction. Thus $\phi = m\texttt{<}h\texttt{>}\ v$ with $m \neq \texttt{private}$ and clauses 5a and 5b hold on M''. Clause 5c holds because $M''_\rho(c) = M'_\rho(c) = m\texttt{<}M'_v(id_f.h)\texttt{>}\ v = m\texttt{<}M''_v(id_f.h)\texttt{>}\ v$. Clause 6 holds because $M''_\rho = M'_\rho$.

A.3.2 Preserving Consistency

Lemma 10 *Single-thread steps preserve memory consistency.* If

$$M, id \models s \quad M, id : s \xrightarrow{s} M'$$

then $id, id' \models M \Rightarrow id, id' \models M'$
Proof: We prove the various aspects of consistency independently, each time by case inspection of the operational semantics' steps.

C1 Remains valid.

C2,3 Trivial for all steps except gcast and scast. We know that $M_\rho(id) = \texttt{private<0>}\ t$, so $M'_\rho(id) = M_\rho(id)$ and $M'_o(id) = M_o(id)$ from the definition of *gcast*. For scast, a private object not in a group can be cast, however, because $id, id' \models M$, we know that id is not the value of any reference that scast could cast, so the invariant still holds for M'.

C4 By inspection, all steps that cause $M'_v(a.f) \neq M_v(a.f)$ define $M'_v(a.f)$ as 0 (null), $c \notin \text{dom}(M)$ (new) or an existing value $M_v(b.g)$. Thus $id \neq 0$, $id \in \text{dom}(M)$ and $\nexists a.f(M_v(a.f) = id)$ imply that $\nexists a.f(M'_v(a.f) = id)$.

C5 Trivial except for assignment steps.

- $x := \ell$, with $lval(M, id, \ell) = a.f$, $c = M_v(a.f)$.

  By Lemma 6, clause 5c is preserved for all variables dependent on x. If $c = 0$ the remaining conditions are trivially true. Otherwise, we must show clauses 5a-c holds for lvalue $id.x$. By assumption

  $$M_\rho(id.x) = m\text{<}x'\text{>}\ t \quad M, id \models \ell : m\text{<}\ell'\text{>}\ t, \omega$$
  $$M, id \models x'[\ell/x] = \ell' \quad lval(M, id, \ell') = b.g$$

  Clause 5a is trivial.

  Clause 5b follows directly from Lemma 5 applied to lvalue ℓ ($m = \texttt{private} \Rightarrow M_o(c) = id$).

  By Lemma 5, $M'_\rho(c) = M_\rho(c) = m\text{<}M_v(b.g)\text{>}\ t$. So, for clause 5c, we must verify that $M'_v(id.x') = M_v(b.g)$. If $x' = x$, $M'_v(id.x') = c$, and $M, id \models \ell = \ell'$ implies $c = M_v(b.g)$. If $x' \neq x$, $M'_v(id.x') = M_v(id.x')$ and $M, id \models x' = \ell'$ implies $M_v(id.x') = M_v(b.g)$.

- $x.f := y$, with $a = M_v(id.x) \neq 0$, $c = M_v(id.y)$.

  By Lemma 6, clause 5c is preserved for all variables dependent on $x.f$. If $c = 0$ the remaining conditions are trivially true. Otherwise, we must show clauses 5a-c holds for lvalue $a.f$. By assumption

  $$M_\rho(id.x) = m'\text{<}x'\text{>}\ v \quad M_\rho(id.y) = m\text{<}y'\text{>}\ t$$
  $$M, id \models x.f : m\text{<}\ell\text{>}\ t, \omega \quad M, id \models \ell[y/x.f] = y'$$
  $$lval(M, id, \ell) = b.g$$

  Clause 5b follows from Lemma 5 applied to lvalue y ($m = \texttt{private} \Rightarrow M_o(c) = id$).

  First we consider the case where $T(t.f) = u$. Then $m = m'$, $\ell = x'$ and $M, id \models x' = y'$. From Lemma 5, $M'_\rho(c) = M_\rho(c) = m\text{<}M_v(id.y')\text{>}\ t$. From clause 4 of memory consistency we can conclude that $a \neq id$, hence $M'_v(id.x') = M_v(id.x') = M_v(id.y')$ and clause 5a holds. Clause 5c is trivial.

  Next we consider the case where $T(t.f) = m\text{<}f\text{>}\ t$. Then $\ell = x.f$ and $M, id \models y = y'$. From Lemma 5, $M'_\rho(c) = M_\rho(c) = m\text{<}M_v(id.y')\text{>}\ t$. As $M'_v(a.f) = M_v(id.y) = M_v(id.y')$, clause 5c holds. Clause 5a is trivial.

  Finally we consider the case where $T(t.f) = m\text{<}g\text{>}\ t$, $f \neq g$. Then $\ell = x.g$ and $M, id \models x.g = y'$. From Lemma 5, $M'_\rho(c) = M_\rho(c) = m\text{<}M_v(id.y')\text{>}\ t$. As $M'_v(a.g) = M_v(a.g) = M_v(id.y')$, clause 5c holds. Clause 5a is trivial.

- $x :=$ new, with $a = M'_v(id.x)$

  By Lemma 6, clause 5c is preserved for all variables dependent on x. We must show clauses 5a-c holds for lvalue $id.x$. Clauses 5a and 5b hold trivially. Clause 5c holds by construction of the $M'_\rho(a)$. Finally, clause 5 holds for the fields of cell a as they are all null.

- x := gcast z, with $M_\rho(id.x) = m\text{<}y\text{>}\ t$, $M_\rho(id.z) = m\text{'<}z\text{>}\ t$, $c = M_v(id.z)$, $M, id \models$ *oneref*(z).

  Let $M'' = M[id.x \xrightarrow{v} c, id.z \xrightarrow{v} 0]$. By two applications of Lemma 6, clause 5c is preserved in M'' for all variables dependent on x and z. If $c = 0$, $M' = M''$ and clause 5 holds. We verify that clause 5 holds for all addresses a when $c \neq 0$:

  Cells outside the group being cast. If $M_\rho(a) = n\text{<}b\text{>}\ u$ with $b \neq c$ and $a \neq id$, then $M(a) = M'(a)$. Consider a field f of a such that $b = M'_v(a.f) \neq 0$. If $M'_\rho(a.f) = v$ then $M_\rho(b) = n\text{<}b\text{>}\ v$. As $b \neq c$, $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$ so clauses 5a and 5b hold. Clause 5c holds trivially (no other fields can be dependent on f). If $M'_\rho(a.f) = n\text{'<}g\text{>}\ v$, then $M_\rho(b) = n\text{'<}M_v(a.g)\text{>}\ v$. *oneref*($z$) implies that $M_v(a.g) \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.

  Cells inside the group being cast. If $M_\rho(a) = n\text{<}c\text{>}\ u$ (this implies $a \neq id$) then $M'_\rho(a) = m\text{<}d\text{>}\ u$, $M'_o(a) = id$, and $\forall f.M'_v(a.f) = M_v(a.f)$ where $d = M''_v(id.y) = M'_v(id.y)$. Consider a field f of a such that $b = M'_v(a.f) \neq 0$. If $M'_\rho(a.f) = v = M_\rho(a.f)$ then $M_\rho(b) = n\text{<}c\text{>}\ v$, so $M'_\rho(b) = m\text{<}d\text{>}\ v$ and $M'_o(b) = id$ so clauses 5a and 5b hold. Clause 5c holds trivially (no other fields can be dependent on f). If $M'_\rho(a.f) = n\text{'<}g\text{>}\ v = M_\rho(a.f)$, then $M_\rho(b) = n\text{'<}M_v(a.g)\text{>}\ v$. *oneref*($z$) implies that $M_v(a.g) \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.

  Local variables of id. Consider a local variable x' of thread id with type $n\text{<}y'\text{>}\ u$.

  - $x' = z$: Clauses 5a-5c hold trivially.
  - $x' = x$: $M_\rho(c) = m\text{'<}c\text{>}\ t$, so $M'_\rho(c) = m\text{<}d\text{>}\ t$ where $d = M''_v(id.y) = M'_v(id.y)$ so clauses 5a-5c hold.
  - $x' \neq x, x' \neq z$: if $b = M'_v(id.x') = M_v(id.x') \neq 0$ then $M_\rho(b) = n\text{<}M_v(id.y')\text{>}\ u$. If $y' = z$, then $M, id \models x' = null$, a contradiction. Otherwise *oneref*(z) implies that $M_v(id.y') \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.

- x := scast z

  Similar to the gcast case, but taking into account the fact that $M, id \models \mathrm{NF}(M, id, z)$, which makes 5a trivial, and simplifies the 5b.

C6 Trivial except for new, gcast, and scast steps, as types are unchanged. For new, the clause holds because STRUCTDEF prohibits explicit `private` fields within structures (this could be relaxed to a weaker rule matching the requirements of clause 6).

For gcast, consider an lvalue $a.f$ with $M'_\rho(a) = m\text{'<}c\text{>}\ t$, $M_\rho(a) = m\text{<}b\text{>}\ t$, $M'_\rho(a.f) = \texttt{private}\text{<}g\text{>}\ t = M_\rho(a.f)$. $id \models M$ implies $b = 0$ and $m = \texttt{private}$, so *gcast* will leave cell a unchanged, i.e. $c = b = 0$ and $m' = m = \texttt{private}$, so clause 6 holds for M'. A similar argument holds for scast.

A.4 Preserving Type Safety

Lemma 11 *Same-thread steps preserve type safety* If

$$id \models M \qquad M, id : s \xrightarrow{s} M'$$

then $M, id \models s; s' \Rightarrow M', id \models s'$
Proof: From Lemma 7, $M'_\rho(id) = M_\rho(id)$, so $M'_\rho(id.x) = M_\rho(id.x)$ for all variables x of thread id. An inspection of the rules show that $M, id, \text{false} \models s'$ depends only on $M_\rho(id)$, and thus $M', id, \text{false} \models s'$, so $M', id \models s'$.

Lemma 12 *Other-thread steps preserve type safety* If

$$id, id' \models M \quad M, id' : s' \xrightarrow{s} M'$$

then $M, id \models s \Rightarrow M', id \models s$
Proof: From Lemma 7, $M'_\rho(id) = M_\rho(id)$, so $M'_\rho(id.x) = M_\rho(id.x)$ for all variables x of thread id. Thus, to show that $M', id \models s$ it suffices to show that already-executed runtime checks are preserved, i.e. $M, id \models \omega \Rightarrow M', id \models \omega$ for all ω that are required by $M, id \models s$. We analyze each check independently:

- $\texttt{private}(x)$ always holds.
- $\texttt{locked}(x)$: By Lemma 8, $a = M'_v(id.x) = M_v(id.x)$, and by assumption $M_L(a) = id$. The only step that could cause $M'_L(a) \neq M_L(a)$ is unlock y with $M_v(id'.y) = a$. However, this can only be executed by id' if $M_L(a) = id'$.
- $\ell = \text{null}$: If $\ell = x$, by Lemma 8, $M'_v(id.x) = M_v(id.x)$ so $M', id \models \ell = \text{null}$. If $\ell = x.f$, by Lemma 8, $a = M'_v(id.x) = M_v(id.x)$. If $M_\rho(a) = \texttt{private}\text{<}b\text{>}\ t$, then, by Lemma 2, $M'_v(a.f) = M_v(a.f)$ so $M', id \models \ell = \text{null}$. If $M_\rho(a) = \texttt{locked}\text{<}b\text{>}\ t$ then from $id \models M$ we know that $M_\rho(id.x) = \texttt{locked}\text{<}y\text{>}\ t$. By inspection of the rules that depend on $\ell = \text{null}$, we note that $M, id \models \texttt{locked}(x)$ must hold, i.e. $M_L(a) = id$. Hence, by Lemma 3, $M'_v(a.f) = M_v(a.f)$ so $M', id \models \ell = \text{null}$
- $\ell = \ell'$: The same logic as for $\ell = \text{null}$ applies.
- *oneref*(x): We know that $id.x$ is the sole lvalue referencing $a = M_v(id.x)$. Only assignment statements by id' could cause this to change. $y = z$, null, new and gcast assignments cannot create or destroy references to a as they modify local variables of $id' \neq id$. $id \models M$ implies that no lvalue references id, so $y.f := z$ cannot modify $id.x$ and similarly that $y := z.f$ cannot create an extra reference to a. Thus $M', id \models \textit{oneref}(x)$.
- *chkread*(x), *chkwrite*(x): By Lemma 8, $a = M'_v(id.x) = M_v(id.x)$, and by assumption $id \in M_W(id.x) \Rightarrow M_R(id.x) \subseteq M_W(id.x) = \{id\}$. Because $M'_v(id.x) = M_v(id.x)$, id' could only have read $id.x$. Therefore, $M_R(id.x) \subseteq M'_R(id.x)$. If $M_W(id.x) = \emptyset$, then we could have $M_R(id.x) \subset M'_R(id.x)$, but the invariant still holds for M' because $id \notin M'_W(id.x)$. If $M_W(id.x) = \{id\}$, then $M_R(id.x) = M'_R(id.x)$, and so the invariant holds for M'.

A.5 Thread Creation and Destruction

These two lemmas show that thread creation and destruction preserve type safety, runtime checks and memory and environmental consistency.

$$\frac{\begin{array}{c}\vdash P \Rightarrow P' \quad 0 \notin \text{dom}(M) \quad g > 0 \quad \text{dom}(M) = \{g\} \oplus (\bigoplus_{f()\{\ldots\} \in P'} \{id_f\}) \oplus (\bigoplus_{x:\tau \in P'} \{o_x\}) \\ S = \{(id_f, \textbf{wait}; s) | f()\{\ldots; s\} \in P'\} \quad M(g) = (\texttt{private<0>}\ t_G, g, 0, v_G, \emptyset, \emptyset) \\ x : \tau \in P' \Rightarrow t_G(x) = \tau \wedge v_G(x) = o_x \wedge M(o_x) = (rtype(M, g, \tau), g, 0, \lambda f.0, \emptyset, \emptyset) \\ f()\{\ldots x : \tau \ldots\} \in P' \Rightarrow M(id_f) = (\texttt{private<0>}\ t_f, id_f, 0, v_f, \emptyset, \emptyset) \wedge \\ t_f(x) = \tau \wedge v_f(x) = 0 \wedge (y : \tau \in P' \Rightarrow t_f(y) = \tau \wedge v_f(y) = o_y)\end{array}}{M, S \oplus \{(g, \text{spawn main}; \textbf{done})\}}$$

Figure A.5: Initial State

Lemma 13 *Thread creation.* If

$$\begin{array}{c} id_f, id \models M \quad M, id_f \models \textbf{wait}; s_f \quad M, id \models s \\ \models M[id_f \xrightarrow{o} 0] \quad id' = \max(\text{dom}(M)) + 1 \\ M' = extend(M, id', id', M_\rho(id_f)) \quad M'' = M'[id' \xrightarrow{v} M_v(id_f)] \end{array}$$

then

$$\begin{array}{c} id_f, id, id' \models M'' \\ M'', id_f \models \textbf{wait}; s_f \quad M'', id \models s \quad M'', id' \models s_f \end{array}$$

Proof: $id_f, id, id' \models M'$ is trivial. Verifying $id_f, id, id' \models M''$ only requires checking that clause 5 holds for all lvalues $id'.y$. Consider $a = M''_v(id'.y) = M_v(id_f.y)$, $M''_\rho(id'.y) = M_\rho(id_f.y) = \phi$. If $a = 0$, then clause 5 holds. Consider $a \neq 0$. If $\phi = v \vee \phi = \texttt{private<}z\texttt{>}\ t$ then $M_o(a) = M_o(id_f)$ and $M_o(a) = M[id_f \xrightarrow{o} 0]_v(id_f) = 0$, a contradiction. Thus $\phi = m\texttt{<}z\texttt{>}\ t$ with $m \neq \texttt{private}$ and clauses 5a and 5b hold on M''. Clause 5c holds because $M''_\rho(a) = M_\rho(a) = m\texttt{<}M_v(id_f.z)\texttt{>}\ t = m\texttt{<}M''_v(id'.z)\texttt{>}\ t$ (note that $a \neq id'$).

$M'', id_f \models \textbf{wait}; s_f$ follows from the fact that $M''_\rho(id_f) = M_\rho(id_f)$, and $M'', id' \models s_f$ follows from the fact that $M''_\rho(id') = M_\rho(id_f)$. Finally, $M'', id \models s$ follows from $M, id \models s$ and the fact that $M, id \models \omega \Rightarrow M'', id \models \omega$: the only interesting case is $oneref(x)$. If $id.x$ is the only reference to cell $a = M_v(id.x)$ then $M_v(id_f.y) \neq a$, so M'' cannot invalidate $oneref(x)$.

Lemma 14 *Thread destruction.* If

$$id, id' \models M \quad M, id \models s \quad M' = M \backslash id'$$

then $id \models M'$ and $M', id \models s$.

Proof: We first show $id \models M'$. Clauses 1-4 follow directly from $id, id' \models M$ and $id \neq id'$. $id, id' \models M$ implies $\nexists a.f(M_v(a.f) = id')$, so clause 5 remains valid for all addresses $b \in \text{dom}(M')$. Clause 6 remains valid as $M'_\rho(b) = M_\rho(b)$ for all addresses $b \in \text{dom}(M')$.

As with thread creation, we note that $M, id \models \omega \Rightarrow M'', id \models \omega$ (easily verified by examining each kind of runtime check, noting that $M_v(id.x) \neq id'$). Furthermore, $M'_\rho(id) = M_\rho(id)$ so $M', id \models s$.

A.6 Soundness Proof

Theorem 6 *Soundness.* Let P be a program with $\vdash P \Rightarrow P'$ and M_0, S_0 be the initial memory and threads for program P' with starting thread main shown in Figure A.5. Let $M, \{(id_1, s_1), \ldots, (id_n, s_n)\}$ be a state reachable in i steps from M_0, S_0. Then

$$id_1, \ldots, id_n \models M \qquad M, id_i \models s_i$$

Proof: We prove the slightly stronger result that the state M, S after step i satisfies

$$\begin{gathered} S = \{(id_{f_1}, \textbf{wait}; s_{f_1}), \ldots, (id_{f_n}, \textbf{wait}; s_{f_n}), (id_1, s_1), \ldots, (id_n, s_n)\} \\ id_{f_1}, \ldots, id_{f_n}, id_1, \ldots, id_n \models M \qquad \models M[id_{f_i} \xrightarrow{o} 0] \\ M, id_{f_i} \models \textbf{wait}; s_{f_i} \qquad M, id_i \models s_i \end{gathered}$$

where the f_i's are the threads declared in P'.

The proof proceeds by induction over the steps of the operational semantics.

By construction, M_0, S_0 satisfies the induction hypothesis.

Assume M, S satisfies the induction hypothesis, and $M, S \rightarrow M', S'$.

First, we handle the prototype threads id_{f_i}. Note that there are no thread step transitions from statements of the form **wait**; s. Thus, $(id_{f_i}, \textbf{wait}; s_{f_i}) \in S'$. Also $M'_\rho(id_{f_i}). = M_\rho(id_{f_i})$ and $M'_v(id_{f_i}) = M_v(id_{f_i})$ (Lemmas 7 and 8), so, by Lemma 9,

$$(id_f \models M') \Rightarrow (\models M'[id_f \xrightarrow{o} 0])$$

To complete the induction we thus only need to show

$$\forall (id, s) \in S'(id \models M') \wedge (M', id \models s)$$

We proceed by analysing each kind of state transition.

- Simple statement:

$$\frac{M, id : s_1 \xrightarrow{s} M'}{M, \{(id, s_1; s_2)\} \oplus S \rightarrow M', \{(id, s_2)\} \cup S}$$

  Let $(id', s') \in S$. By induction, $id, id' \models M$, $M, id \models s_1; s_2$ and $M, id' \models s'$. By Lemma 11 $M', id \models s_2$, by Lemma 12 $M', id' \models s'$ and by Lemma 10, $id, id' \models M'$. Lemma 1 completes the induction for this case.

- Runtime check:

$$\frac{M, id \models \omega_1 \qquad \Omega = \omega_2, \ldots, \omega_n}{M, \{(id, \ell := e \text{ when } \omega_1, \Omega; s)\} \oplus S \rightarrow M, \{(id, \ell := e \text{ when } \Omega; s)\} \cup S}$$

  We only need to verify that

$$M, id \models id, \ell := e \text{ when } \omega_2 \ldots, \omega_n; s$$

  Examination of the assignment rules shows that this holds because $M, id \models \omega_1$.

- Thread creation: The induction follows from Lemmas 13 and 1.
- Thread destruction: The induction follows from Lemmas 14 and 1.

Theorem 7 *Safety of* `private` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{private<}b\texttt{>}\ t$ then $M_o(a) = id$.
Proof: From Theorem 6 and Lemma 2.

Theorem 8 *Safety of* `locked` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{locked<}b\texttt{>}\ t$ then $M_L(a) = id$.
Proof: From Theorem 6 and Lemma 3.

Theorem 9 *Safety of* `dynamic` *accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{dynamic<}b\texttt{>}\ t$ then $id \in M_W(a) \Rightarrow M_R(a) \subseteq M_W(a) = \{id\}$. **Proof:** From Theorem 6 and Lemma 4.

Appendix B

The Soundness of Shelters

THEOREM 1. *Atomicity.* For every valid trace T with results M there exists a serial trace T' with results M. Furthermore, atomicorder(T) = atomicorder(T') and for every thread t, ops(t, T) = ops(t, T').

Proof: The proof proceeds by a step-wise transformation of T into a serial trace T'. Any non-serial trace T has a (possibly empty) prefix that matches a serial trace, i.e. T is of the form:

$$\begin{array}{l}(t^1, \text{atomic}_{n^1}(\ldots)), (t^1, v^1_1 := \ldots), \ldots, (t^1, v^1_{m^1} := \ldots), (t^1, \text{endatomic}) \\ \ldots \\ (t^k, \text{atomic}_{n^k}(\ldots)), (t^k, v^k_1 := \ldots), \ldots, (t^k, v^k_{m^k} := \ldots), (t^k, \text{endatomic}), \\ (t_1, s_1), \ldots, (t_n, s_n)\end{array}$$

Furthermore s_1 must be an atomic statement, as assignments and endatomic are not valid when no atomic statement is in progress. Therefore $(t_1, s_1), \ldots, (t_n, s_n)$ must match the following template:

$$\begin{array}{l}(t, \text{atomic}_{n_t}(\ldots), (t, v_1 := \ldots), \ldots, (t, v_{m_t} := \ldots), \\ (t', s'), \ldots, (t'', s''), (t, s), \ldots\end{array}$$

with $n_t > n^k$, $t' \neq t$ and (t, s) the earliest statement of thread t after $v_{m_t} := \ldots$: there must be at least one such statement as a valid trace requires $A(t) = 0$ at termination so thread t must have at least one more endatomic statement. Note also that s cannot be an atomic statement.

We show that the trace T'' produced by moving (t, s) one step left, i.e. swapping it with (t'', s'') is a valid trace with the same results M, atomicorder(T) = atomicorder)(T'') and $\forall t.\text{ops}(t, T) = \text{ops}(t, T'')$. Repeated applications of this transformation clearly terminate in the desired serial trace T': we eventually move (t, s) until it is adjacent to $(t, v_{m_t} := \ldots)$. At that point we will start moving another statement left, either for thread t if $s \neq$ endatomic or for some new statement for thread t'. We note that this approach is very similar, but not quite identical to Lipton's theory of reduction [63]: while in our particular case $(t, v := \ldots)$ can be moved left past $(t'', \text{endatomic})$ (see below), in general assignments cannot be moved past endatomic statements of other threads, i.e. are not general left-movers.

We show that moving (t, s) left produces T'' with the desired properties for the six possible combinations of statements s'' and s. We start by noting that as s is not an atomic statement and $t'' \neq t$ that atomicorder(T) = atomicorder)(T'') and $\forall t.\text{ops}(t, T) = \text{ops}(t, T'')$. For each case we

only need to show that the trace T'' is valid and has results M. We denote the operational state of T before (t'', s'') by M_0, A_0, a_0, Σ_0, the state after (t'', s'') by M_1, A_1, a_1, Σ_1 and the state after (t, s) by M_2, A_2, a_2, Σ_2. The state in T'' before (t, s) is also M_0, A_0, a_0, Σ_0, after (t, s) it is $M''_1, A''_1, a''_1, \Sigma''_1$ and the state after (t'', s'') it is $M''_2, A''_2, a''_2, \Sigma''_2$. We simply need to show that $(M''_2, A''_2, a''_2, \Sigma''_2) = (M_2, A_2, a_2, \Sigma_2)$.

First, we consider the case where s is endatomic. If s'' is endatomic or s'' is atomic the result follows from the obvious lemma that $u \neq u' \wedge A(u) \neq 0 \Rightarrow A(u) \neq A(u')$ applied to $t \neq t''$. The case where s'' is $v'' := \ldots$ also holds, as if $A_0(t'')$ is the smallest value in $\Sigma_0(v''_\sigma) \cup \Sigma_0(G(v''))$ it is also clearly the smallest value in $\Sigma''_1(v''_\sigma) \cup \Sigma''_1(G(v''))$ (the same argument applies to any other variable in the assignment).

Next, we consider the case where s is $v := \ldots$. We know that $A_0(t) = A_1(t) = n_t$. If s'' is atomic$_p(\ldots)$, then $p > n_t$ so access$(A_1(t), v, \Sigma_1) \Rightarrow$ access$(A_0(t), v, \Sigma_0)$ so the result clearly holds. If s'' is $v'' := \ldots$ and the two statements access non-overlapping variables, the result is trivial. The case where they access overlapping variables cannot occur as T would not be a valid trace: $A_0 = A_1$ and $\Sigma_0 = \Sigma_1$, so if access$(A_1(t), v, \Sigma_1)$ then $\neg$access$(A_0(t''), v, \Sigma_0)$, or vice-versa. If s'' is endatomic we note that $A_0(t'') > A_0(t) = A_1(t)$ as t'' must have started its atomic statement after t (all atomic statements prior to t's have already completed). Thus access$(A_1(t), v, \Sigma_1) \Rightarrow$ access$(A_0(t), v, \Sigma_0)$ so the result clearly holds.

Appendix C

Concurrent Reference Counting

SharC builds upon the Heapsafe system [43] for reference counting C. Applying this work directly in SharC implies atomically updating reference counts for all pointer writes. The resulting overhead is unacceptable on current hardware, even on x86 family processors that support atomic increment and decrement instructions. To reduce this overhead, SharC can optionally perform a straightforward whole-program, flow-insensitive, type-qualifier-like analysis to detect which locations might be subject to a sharing cast. Only pointers to these locations need reference count updates. However, even with this optimization, the runtime overhead is still too high (over 60% in many cases). To reduce this overhead, we adapted Levanoni and Petrank's high performance concurrent reference counting algorithm [62] (designed for garbage collection) for use with SharC.

C.1 Overview

In Levanoni and Petrank's algorithm, each thread keeps a local unsynchronized log of the references that have had their values overwritten by that thread. To keep the log small, an entry is only added the first time a location is overwritten since the last collection. A *dirty* bit associated with each location is used to record whether a mutator thread has previously overwritten that location. To bring all these logs together, a dedicated coordinator thread periodically stops all the mutators, grabs the logs they kept of their mutations, clears the dirty bits, sets the threads running again, and then computes updated reference counts by processing the merged logs. To compute the updated reference counts, the coordinator walks through the grabbed logs, and for each entry, decrements the overwritten references, and increments the reference counts for the references currently in the reference cells. It is necessary to take into account mutations since the threads were restarted. For these, if a reference cell is dirty when the coordinator wants to increment the reference count for its current value, it instead finds the recently overwritten reference value in the live update logs, and increments the reference count for that reference. Although this does not reflect the *current* reference counts, it reflects the reference counts that were correct at that snapshotted time. Levanoni and Petrank also have another algorithm that stops threads one by one, rather than all at once, but which is more complicated to implement.

We have adapted Levanoni and Petrank's simpler algorithm to avoid the need to stop all threads while the coordinator grabs the buffers and clears the dirty bits. In our algorithm, there are two sets

```
1  void update(void **slot, void *new) {
2    void *old = *slot;
3    choosing[tid] = 1;
4    dummyCall();
5    rcidxs[tid] = rcidx;
6    choosing[tid] = 0;
7    if (notDirty(rcidxs[tid],slot)) {
8      *log[rcidxs[tid]][tid]++ = (slot,old);
9      markDirty(rcidxs[tid],slot);
10   }
11   rcidxs[tid] = -1;
12   *slot = new;
13 }
```

Figure C.1: The procedure for updating a reference.

of logs and two sets of dirty bits. When the collector thread wants to obtain an accurate snapshot view of the reference counts, it makes all the threads switch to their other update log and the other set of dirty bits (which the coordinator has just cleared). A simple non-blocking algorithm is used to ensure that all threads switch over together. The coordinating thread need only pause as long as it takes the mutator threads to complete any outstanding writes to their logs.

C.2 Reference Update

The procedure for updating a reference is given in Figure C.1. The goal of this procedure, aside from updating the thread-private log, is to ensure that any thread trying to calculate reference counts knows which set of logs the mutating threads are using. First, the old reference is recorded before the dirty bit is inspected. This is so that a race on the dirty bit will at worst result in a duplicate log entry. Next, we set a flag that indicates we are recording which set of logs this update will use. The reasoning here is similar to that in Lamport's well-known bakery algorithm for mutual exclusion. The coordinating thread may not begin computing while there are threads deciding which set of logs to use since they might decide to use the old set.

The next step is to record the set of logs being used for the update by loading the global variable `rcidx` and storing it into an array that records the logs currently being used by each thread. In the absence of sequential consistency, it is possible that a processor may reorder the load of `rcidx` before the store to `choosing[tid]`. This is a problem since it could result in a thread performing an update using the old set of logs.

In our algorithm we rely only on the guarantees specified by Intel for the x86 architecture[55]. This says that, (1) loads are not reordered with other loads and stores are not reordered with other stores, (2) stores are not reordered with older loads, *but* (3) loads **may** be reordered with older stores to different locations. Fortunately, we can work around point 3 by inserting additional loads and stores to the same location, preventing undesired reordering. The call on line 4 is to a function that does nothing but return. In pushing the return address onto the stack and then popping it off,

it performs the needed additional loads and stores, and prevents the processor from reordering the read of `rcidx` before the store to `choosing[tid]`.

After choosing a set of logs, if the reference cell is not dirty, the old value is logged, and the cell is marked dirty. The thread's entry in `rcidxs` is set to -1 to indicate that it is no longer using any set of logs, and finally the update is performed.

C.3 Reference Count Calculation

The procedure for calculating a snapshot of the reference counts is given in Figure C.2. A mutex is acquired so that only one thread may be acting as coordinator at a time. Next, the old logs are swapped out, and then we wait for each of the mutators to start using the new logs. Starvation is impossible since the store to `rcidx` will eventually be visible to all threads. Next, the old logs are merged, and for each of the entries, the reference count for the old value is decremented. If the reference cell has not been updated according to the new logs, the reference count for the new value is incremented. If the reference cell has been updated according to the new logs, we add the cell to a set of "Undetermined" cells. Then, we merge the new logs, and for each entry, if the reference cell for the new entry is in the Undetermined set, the reference count for the old value is incremented. The reference count for the new value will be underestimated, but none of these underestimates will be of any consequence: before calculating a reference count, the reference in question is copied into a local variable and then nulled out. If, during the reference count, another thread is updating a reference cell with the same location we are reference counting, then there must have been more than one reference to the location to begin with. So, we will sometimes underestimate the reference count by one, but only when it would have been three or larger. The bad cast will be detected whether or not there is an underestimate.

```
1  int existsOldOrChoosing(int idx) {
2    int i;
3    for (i = 0; i < MAXTID; i++)
4      if (choosing[i] OR rcidxs[i] == idx)
5        return 1;
6    return 0;
7  }
8
9  void updateRefCounts(void *p, int sz) {
10   int oldidx, cnt = 0;
11   void *end = p + sz;
12   Log ← ∅; Undet ← ∅;
13   mutexLock(refcntLock);
14
15   oldidx = rcidx;
16   rcidx = rcidx ?  0 :  1;
17   while (existsOldOrChoosing(oldidx))
18     yield();
19
20   foreach tid ∈ Threads
21     Log ← Log ∪ log[oldidx][tid];
22   foreach (slot,old) ∈ Log
23     void *new = *slot;
24     refcounts[old]--;
25     if (notDirty(rcidx,slot)) refcounts[new]++;
26     else Undet ← Undet + slot;
27   Log ← ∅;
28   foreach tid ∈ Threads
29     Log ← Log ∪ log[rcidx][tid];
30   foreach (slot,old) ∈ Log
31     if (slot ∈ Undet)
32       refcounts[old]++;
33
34   while (p < end) cnt += refcounts[p++];
35   mutexUnlock(refcntLock);
36 }
```

Figure C.2: The procedure for calculating reference counts.